NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

AUTOMATING THE SIGN-OUT SHEET AS A PART OF THE TACTICAL COMBAT TRAINING PROGRAM AS AN INFORMATION SYSTEM FOR COMMANDING PERSONNEL IN A NAVAL AIRSTATION

by

Enno F. Busch

September 1999

Thesis Advisor: C. Thomas Wu
Second Reader: Michael Capps

Approved for public release; distribution is unlimited.

REPORT DOCUMENTATION PAGE

Form Approved OMB No. 0704-0188

1. AGENCY USE ONLY ***(Leave blank)***	**2. REPORT DATE** September 1999	**3. REPORT TYPE AND DATES COVERED** Master's Thesis

4. TITLE AND SUBTITLE: AUTOMATING THE SIGN-OUT SHEET AS A PART OF THE TACTICAL COMBAT TRAINING PROGRAM AS AN INFORMATION SYSTEM FOR COMMANDING PERSONNEL IN A NAVAL AIRSTATION

5. FUNDING NUMBERS

6. AUTHOR(S)
Busch, Enno F.

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Naval Postgraduate School
Monterey, CA 93943-5000

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

10. SPONSORING / MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

12a. DISTRIBUTION / AVAILABILITY STATEMENT

Approved for public release; distribution is unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT : Presently, the German Navy Airwings lacks the automated computer infrastructure required to process administrative flying specific data in the squadrons and administer data from deployed aircrews on foreign airfields. Thus, flying data are obsolete quickly, especially if aircrews are deployed, and administering personnel must create the forms. Owing to this inefficiency, delays of several days can occur. In this thesis, a prototype of a database and a client-server application was designed and developed showing the feasibility of implementing a system that transfers data quickly and securely over long distance and can store the data accordingly.

With several technologies available to create and implement such a system, the goal was to employ currently available components efficiently and economically. Thus, three-tier client-server implementations are introduced and compared, where the RMI network protocol best provides a robust and efficient answer for the required needs. Java RMI enables the software developer to create distributed Java-to-Java applications in which the methods of remote Java objects can be invoked from objects in other Java Virtual Machines (JVM). RMI can be programmed to provide services to a database by establishing a listener process to handle access requests. RMI uses the built-in Java security mechanisms, making the computer system safe when code is downloaded. Every RMI-based system is portable to any JVM. The system incorporates the JDBC™ to establish a connection to a database.

14. SUBJECT TERMS: Tactical Combat Training Program, RMI, Java, JVM, Servlets, Client-Server, JDBC™.

15. NUMBER OF PAGES
118

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT Unclassified	**18. SECURITY CLASSIFICATION OF THIS PAGE** Unclassified	**19. SECURITY CLASSIFI- CATION OF ABSTRACT** Unclassified	**20. LIMITATION OF ABSTRACT** UL

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18

Approved for public release; distribution is unlimited

AUTOMATING THE SIGN-OUT SHEET AS A PART OF THE TACTICAL COMBAT TRAINING PROGRAM AS AN INFORMATION SYSTEM FOR COMMANDING PERSONNEL IN A NAVAL AIRSTATION

Busch, Enno F.
Commander, German Navy

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
September 1999

Author: ______________________
Enno F. Busch

Approved by: ______________________
C. Thomas Wu, Thesis Advisor

____
Michael Capps, Second Reader

____
Dan Boger, Chairman
Department of Computer Science

ABSTRACT

Presently, the German Navy Airwings lacks the automated computer infrastructure required to process administrative flying specific data in the squadrons and administer data from deployed aircrews on foreign airfields. Thus, flying data are obsolete quickly, especially if aircrews are deployed, and administering personnel must create the forms. Owing to this inefficiency, delays of several days can occur. In this thesis, a prototype of a database and a client-server application was designed and developed showing the feasibility of implementing a system that transfers data quickly and securely over long distance and can store the data accordingly.

With several technologies available to create and implement such a system, the goal was to employ currently available components efficiently and economically. Thus, three-tier client-server implementations are introduced and compared, where the RMI network protocol best provides a robust and efficient answer for the required needs. Java RMI enables the software developer to create distributed Java-to-Java applications in which the methods of remote Java objects can be invoked from objects in other Java Virtual Machines (JVM). RMI can be programmed to provide services to a database by establishing a listener process to handle access requests. RMI uses the built-in Java security mechanisms, making the computer system safe when code is downloaded. Every RMI-based system is portable to any JVM. The system incorporates the JDBC™ to establish a connection to a database.

TABLE OF CONTENTS

LIST OF FIGURES

ACKNOWLEDGEMENT

…to Nicole, Kim, Simon, and Fabian: Thank you for your patience and your understanding and for reminding me from time to time, that there is something more besides computers.

I. INTRODUCTION

A. BACKGROUND

Presently, the German Navy Airwings lacks the automated computer infrastructure required to process efficiently administrative flying specific data in its squadrons and deployed airfields. This impedes the leadership's ability to obtain an overall view of flying hours for aircrews and flying events. As a result, administrative flying data are out-of-date very quickly, especially if aircrews are deployed to foreign countries. In this case, special forms, called "sign-out sheets," are collected at the foreign airfield, and administrative personnel must make up the surveys after gathering these sign-out sheets. Sometimes, a delay of days or even several weeks occurs. Furthermore, the lack of an adequate computerized support system results in redundant and imprecise flight data storage and administration. This inefficient method wastes manpower and time and also decreases accuracy.

Previous attempts to implement a more accurate system in an airwing only forced each squadron to meet its own requirements crudely, by using a simple spreadsheet. A standardized and satisfactory solution for the airwing was never found. Furthermore, it was never possible to connect to the home base and send specific flying data to administrative staff personnel. However, borrowing from these earlier attempts, I designed and built a prototype of a database and a client/server application to show the feasibility of implementing a system that can be used for transferring data quickly and securely over long distance.

Due to downsizing and immense budget cuts, the German Navy is required to save money wherever possible and must implement existing technical components that are presently available and still usable. With these restrictions in mind, I used advanced computer network technology to connect a database to the network. By implementing existing Commercial-Off-The-Shelf (COTS) products with existing components, the designed computer system will be economical and efficient. From this research a prototype will be designed and built on an open network Client/Server environment. The objective of this thesis is to provide access to a database at an airwing from the outside via common modern technology. There are two reasons for building such a system:

- Commanding personnel in an airwing will have immediate access to the number of missions, flying hours and flying events conducted. Survey results will be immediately available for analysis and further use.
- The implementation of the system can reduce working hours by preventing double entries of data from flying-crews and personnel who administer the sign-out sheets. This in turn will reduce the possibility of human error.

B. CHALLENGE

The purpose of this thesis is to specify the requirements for implementing a part of the Tactical Combat Training Program (TCTP) and to automate this part by designing a database, and a Graphical User Interface (GUI), and after analyzing some alternatives, to select the appropriate architecture in order to implement the necessary connections to the squadrons and to foreign airfields via the Internet. The goal is to replace the current manual efforts for aircrews and administrators and to create an automated system superior

to the "sign-out" sheets. To achieve this goal, special software, called *Middleware*, which connects two otherwise separate programs will be implemented. The term *Middleware* is used to explain a program that serves as a connection between two applications and passes data between them. In a three-tier architecture, which is explained in Chapter VI, *Middleware* engages the middle-tier.

C. THESIS ORGANIZATION

The remainder of this thesis is organized into the following chapters:

- Chapter II: Tactical Combat Training Program. This chapter describes parts of the Tactical Combat Training Program (TCTP) that are related to the thesis and provides a brief introduction of the present workflow. The chapter also shows the need to create an automated system.
- Chapter III: System Design Constraints. This chapter describes the requirements for the planned implementation of the system, and explains the design constraints for the future implementation provided as a guideline by the present workflow in the squadrons.
- Chapter IV: Relationship with Previous Work. This chapter describes the approaches of other theses and how similar problems were previously solved, and why it was infeasible to build on these earlier approaches. This chapter also shows technical constraints that lead to the approach demonstrated in this thesis.
- Chapter V: Java and JDBC™ This chapter describes the advantages of the JDBC™ database access from Java programs. JDBC™ is a trademark name

and not an acronym. However, quite often it is thought of as Java-DataBase-Connectivity [SUND98]. This chapter also highlights the distinctive Java and JDBC™ features for efficiently manipulating data.

- Chapter VI: Connections to the Clients and to the Internet This chapter describes the different examples of the client-server architectures.
- Chapter VII: Discussion of RMI and Servlets This chapter provides an overview of RMI and Java servlets and compares a Multi-Tier policy with *Remote-Method-Invocation* (RMI) by examining and describing the similarities and the differences of both architectures. First, the RMI framework is discussed as far as architecture, client interfaces, and security are concerned. Second, Java servlets are introduced and compared to the predecessor, the Common Gateway Interface (GCI).
- Chapter VIII: System Development and Implementation This chapter explains the reasons and the advantages of the system that led to the decision for implementing a particular network protocol. It describes and explains the stages of the development and the reasons for product implementation. It also describes the implemented classes for the prototype and the operation of the designed system.
- Chapter IX: Conclusion and Future Research This chapter briefly summarizes the present situation regarding administering the sign-out sheet, the benefits of implementing the component-based client-server solution *RMI,* that uses the Internet, and recommends how to enhance the introduced prototype in the near future.

II. TACTICAL COMBAT TRAINING PROGRAM

Headquarters and units must be able to accomplish their assigned mission in peace time as well as during a crisis, a conflict, or during a war. In order to achieve a high standard of readiness for action, continuous training must be enforced. Therefore, special regulations for training and evaluation are established. The general requirements of the Tactical Combat Training Program (TCTP) are stated in Air Forces Standard (AFS) and include criteria and guidance for the aircrews' flying time and flying events. For flying headquarters, like the Flotilla Naval Air Arm, and their associated units, the TCTP is the basis for the annual planning of the distribution of flying sorties and the basis for planning the distribution of tactical training events. For efficiency reasons, the flying crews have to perform as many events as possible during one flying sortie.

The basis for the TCTP is the Air Forces Standard (AFS), which provides the guidelines for achieving the TCTP planning. Aircrews have to fly a specific amount of flying hours and have to fulfill a specific amount of events in order to keep their *Combat Ready* or *Limited Combat Ready* status. Furthermore, all crew members must pass an annual flying performance check to prove their skills and to maintain their flying rating.

Part of the TCTP is a particular form, the "sign out sheet." After each flight, aircrews are required to fill out the "sign-out sheet" to record their flying time and to administer their flying events for the mission. It is more necessary than ever that units be capable of deploying to and operating from locations other than their main operating installation or peacetime location. Despite the deployment of units from their home location, they must also be able to establish and maintain a close contact and must be

capable to contribute important information across long distances in almost real time. Hence, the instantaneous connection from a deployed operating base becomes significant for transmitting information between the personnel in headquarters, an airstation and the flying crews in foreign countries.

III. SYSTEM DESIGN CONSTRAINTS

A. GENERAL REQUIREMENTS

The accomplishment of this computer system should allow for immediate notification of appropriate personnel and should also save time for administering the sign-out sheets. The computer system uses common technology for data sources, but uses a newer technology in the form of hardware connectivity to the outside world for fast information update. The system is tailored to the needs and preferences of a Naval Airwing and the system represents the information in an easily readable format. Usually, these information systems were designed as mainframe computer-based systems with access from several workstations. The intent was to offer key information for decision-makers, who are not very familiar with the computer environment. Due to cost reduction in the forces, developing and implementing a system that minimized costs was important. Also a purchase of significant hardware had to be avoided. The computer system takes advantage of the client-server technology so that each workstation in the squadrons and deployed airfield has access to the stored information at the home base. Therefore, on base, the system will operate over the traditional network system, and from foreign airfields access to the home base server is established via the Internet.

B. DISTRIBUTED SYSTEM

The term *distributed system* is used to define a model for the interaction between concurrently operating software processes. The computer system consists of several

deployed computers (clients) not sharing a common memory. The client processes send requests to a server process, and the server responds to those requests. The communication between the clients and the server processes is a cooperative exchange in which the client is the proactive part and the server the reactive part. As such, both the client and the server communicate by sending and receiving messages across a complex network [SISH94]. In general, the client computers are single-user PCs, often notebooks, or workstations and present a sophisticated, but a user-friendly interface to the personnel. The most common type of server activity is the database server, containing the *Data-Base-Management-System* (DBMS) with its data. Currently, the most common type of DBMS is the *relational database*. This server enables many clients to share the access to the same database and enables a high-performance computer system to manage the database. Furthermore, an important entity in the system is the network. Users, applications, and resources are distributed according the requirements, and they are linked by the *Internet* or other *Wide-Area-Network* (WAN).

C. WORKSTATION MODEL

The *Workstation Model* is one of the three categories for classifying distributed systems [TARE85]. In this model, the computer system consists of several workstations that are spatially distributed. Each person has his or her own workstation, the client machine, on which the user's application is performed. The client machine is responsible only for providing the GUI; all the processing work is done on a server machine. This configuration ensures that the client is best suited for providing the user-interface and that all applications, like the databases, can be easily maintained on the remote central-system.

In this way, the client can access the information stored on a database regardless of the location of the database or the location of the user's computer. This is done through an associated *File-System*. The primary benefit of this model is that new hardware can be easily added without interrupting the operation while one is running the system. Also, reliability and availability is improved because whenever a component of the system fails, the rest of the computer system will not be affected [TANE97].

D. COMMUNICATION ARCHITECTURE

All computers in a distributed system are connected through a computer network. The communication architecture is special software that supports the network of distributed and independent client computers. The software contributes support for distributed applications, such as remote computer access or data transfer through the network. Nevertheless, each computer preserves a distinct client identity and preserves a specific application that must be communicated to other computers by explicit references. Moreover, each computer has its own operating system, and thus a heterogeneous mixture of computers and operating systems is possible, as long as all the computers possess the same architecture.

THIS PAGE INTENTIONALLY LEFT BLANK

IV. RELATIONSHIP WITH PREVIOUS WORK

Middleware that handles interactions between software applications, and the operating system and the network services is not new. *Middleware* is the connecting software that allows multiple processes running on one or more computers to interact across a network. For instance, for years a client machine has been able to invoke a method on a server object when both were on the same machine, or it could invoke a method across the network using a variety of communication protocols. However, the method of connecting a database to the clients via a server by using the RMI framework, is rather unique in design and provides several advantages over other distributed computing alternatives, such as *Common-Object-Request-Broker-Architecture* (CORBA) or *Caffeine*.

CORBA is an open standard-based software solution for distributed computing. The primary advantage of using CORBA is that clients and servers can communicate in any programming language [AL99]. There is no need to predetermine the programming language, the hardware platform, the operating system, or the degree of application distribution, such as when the application runs locally or remotely. Certainly, this increases the flexibility for application architectures, as well as simplifying and clarifying the distributed computing environment, since the element roles are dynamic. This means an object can either behave as a client, or it can present service to other participants in the network, acting as a server. LT. Akbay and LT. Lewis' thesis outlined how to design and build a distributed application by using CORBA [AL99].

The method of connecting a database with a client by using RMI is one method that allows the software developer to produce distributed Java-to-Java applications, in which the methods of remote Java objects can be called from other Java virtual machines (JVM). Like CORBA, the RMI framework connects remote objects to each other, sending objects and invoking methods regardless of what machine they are on. Presently, RMI is usable for all-Java implementations, and CORBA is applicable for a mixture of Java and other applications written in *different* languages. However, some industrial work is under way to integrate CORBA and RMI into a single and coherent standard.

LT. Held and CDR. Mingo, NPS Monterey, examined the administration of an Internet-based model of an airsafety survey [HM99]. This thesis describes the project development through the research process and further describes the project design, including the informational flow of data. Held and Mingo's thesis also includes Internet technologies, such as *Active Server Pages, HTML, Active X, CGI Scripts and Secure Socket Layer.* This comprehensive thesis additionally includes various important technologies to build a distributed computer system, including an automated survey analysis.

Fortunately, for the realization of a project to allow a connection to foreign airfields and to allow, consequently, the automated data transmission across the Internet and across a military installation, a simpler protocol is sufficient in order to achieve the necessary requirements.

The RMI framework is a relatively simple protocol, but unlike the more complex protocols, such as CORBA, it works only with Java objects. As such, RMI is the perfect application framework that allows efficient interoperability between objects running on

different JVMs and provides even more advantages; therefore, I decided to design and to create a Java-based client-server application for the predefined needs of the Naval Airwing. As a future step, more sophisticated technologies, like applets and servlets, should be designed and implemented in order to execute an applet by a Java-enabled Web browser on the client computer, as well as to improve the system's performance and the security.

THIS PAGE INTENTIONALLY LEFT BLANK

V. JAVA AND JDBC™

A. INTRODUCTION

The JDBC™ API is an interface based on Open-Data-Base-Connectivity (ODBC). The ODBC API is well-established and is the most widely distributed programming interface that accesses a relational database. This system provides the ability to connect to several databases on almost every platform. ODBC manages the access to the DBMS by inserting a middlelayer between the application and the data-retrieval-system. The purpose of the middlelayer is to convert the database queries into a syntax that the DBMS understands. In order to accomplish this goal, the application and the DBMS must be ODBC compliant. This means the application must be able to provide ODBC commands, and the DBMS must be able to reply to these commands accordingly.

Yet, using ODBC directly from the Java language to access the database is not feasible. The connection is best done by using the JDBC™ API in form of a JDBC™-ODBC bridge. JDBC™ consists of a set of classes and interfaces written in Java that make database access from a Java environment easy and powerful. ODBC and JDBC™, both APIs, provide similar functionality, but they are very different in their implementation.

The reason for implementing the JDBC™ is manifold. The most obvious difference is that ODBC is written in the C language and uses pointers. Since Java does not use pointers, transferring the code is extremely difficult. ODBC uses a C language interface; hence, program calls from the Java language to the C language have various

drawbacks, such as security, robustness, and the automatic portability of programs. Second, the design goals of the APIs are different. JDBC™ was designed to be very compact and to act as a simple interface focusing on the execution of SQL statements and retrieving the results. Contrarily, ODBC is much larger in scope and attempts to pack as many features as possible in each driver. The ODBC API combines simple and complex features. Therefore, the added functionality of ODBC is sometimes very useful, but can make even simple tasks difficult and ineffective to perform. For a "pure Java" implementation, a Java API such as JDBC™ is essential. When just using the ODBC driver manager, all drivers have to be installed by hand on each client. But the JDBC™ driver is written in the Java language; therefore, the JDBC™ code can be automatically installed and is automatically secured on all Java platforms. As a consequence, the JDBC™ API is a Java interface to the basic SQL queries and concepts. JDBC™ builds on ODBC and retains the basic architectural features of ODBC. Furthermore, JDBC™ continues to develop on the ODBC foundations and continues to reinforce the Java style.

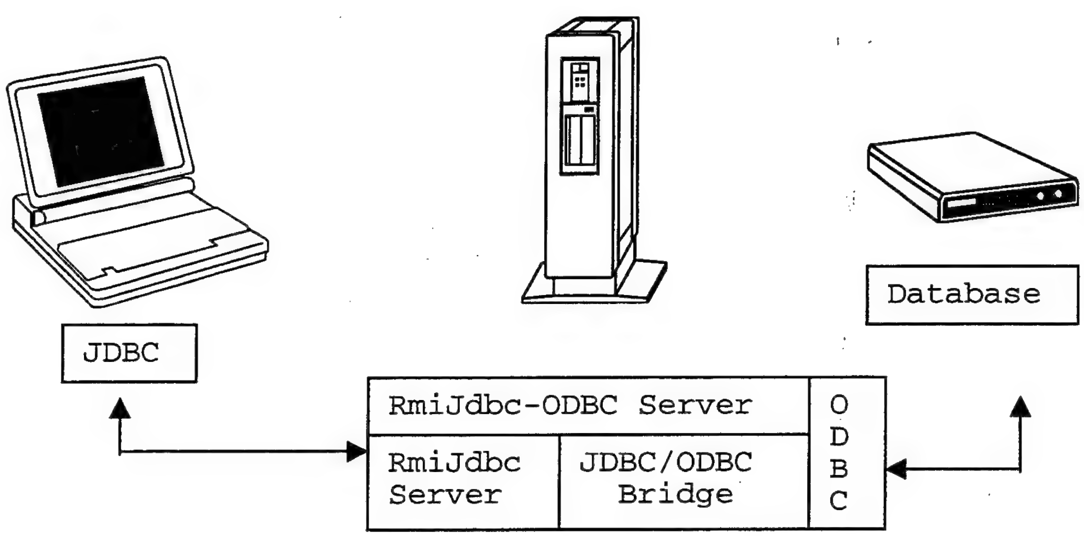


Figure 5.1: RmiJdbc and ODBC Server Component

The next two sections introduce the Java language and the JDBC™ API in detail.

B. JAVA

Java is an object-oriented programming language used to create applications and applets. Java applications are stand-alone programs, which do not require a Web browser. Java applets add executable content to Web pages. Furthermore, Java applets can employ animation, interaction and video imaging.

Java has numerous advantages over other object-oriented languages. The most important Java features are [HCF97]:

- **Portability:** Java code is compiled into platform neutral byte-code and is able to run on any O.S. that provides a Java interpreter. The Java compiler creates the byte-code that is interpreted by the Java Virtual Machine (JVM) at run-time. Therefore, any application or applet written in Java is platform independent and any computer equipped with a Java capable Web browser can run these Java programs.
- **Object-Orientation:** The Java language is a pure object-oriented language. All primitive data types, like *boolean*, *int*, *float*, *character*, and *double* have class wrappers. Wrapper classes make it possible to turn a primitive value into an object so that the value can be passed to a generic class or method, which requires an object reference.
- **Security:** The JVM was designed to maximize security. Java measures provide trustworthy applets and ensure that applications are implemented

according to specified security rules. The implemented byte-code checker makes sure that the code is correct (does nor forge pointers), does not violate access restrictions, and uses objects only for what they are intended.

- **Simplicity:** The Java language syntax is similar to C++. In contrast to C++ Java supports garbage collection. Whenever an object is no longer being used, Java automatically removes it from the memory. Hence, no memory management is necessary.
- **Threads:** Java provides the ability to control several threads at the same time. The ability to execute threads concurrently is required for animation programs and networking.
- **GUI Support:** The Java programming language supports easy-to-create GUI programs with many window components.
- **Dynamic Incremental Loading and Linking:** Java classes are linked dynamically at load time. Thus, adding new methods and data fields to a class does not require recompilation of client classes. Additionally, applications can execute statements to look up fields and methods and then use them accordingly.
- **Internationalization:** All Java programs are written in Unicode. This is a 16-bit character code that embodies the alphabets from the most widely used languages around the world. Software developing personnel appreciate the Java language for its ability to manipulate unicode characters and to support local time and dates.

- **Network:** The Java programming language is designed for network communication, Web applets, and for client-server applications. Moreover, it is ideal for remote access to databases, to programs and to methods.

Clearly, there are many reasons why Java is so popular and useful. Aside from the above-mentioned items, the most important features of the Java language are demonstrated in this thesis. In the next section, the JDBC™ application programming interface (API) and how JDBC™ is related to the Java language, and how it is related to the Structured Query Language (SQL) is introduced. Incidentally, JDBC™, neither an acronym nor an abbreviation, is considered a legal trademark.

C. JDBC™

There are many different database models and standards for accessing data retrieval systems. The Java Database Connectivity Standard is platform-independent and database-independent for talking to an SQL data retrieval system. The JDBC™ API is used for connecting Java to databases and performing transactions (SQL statements) with the database. The JDBC™ consists of several classes and interfaces, all written in Java, and has a standard API included that permits it to write database applications using pure Java API. By using the JDBC™ API, the back-end database can be changed without major modifications. Any program written with JDBC™ can transfer SQL statements to different database architectures simultaneously. This enables administrative personnel to continue using an old-fashioned database, as well as the modern and sophisticated client/server computer systems.

The JDBC™ can perform the following steps to access a database:

1. Importing the Java.sql.package
2. Loading and registering the driver
3. Establishing a connection to a database
4. Sending SQL statements
5. Executing of a statement
6. Retrieving the results
7. Closing the connection and statement

Example Code:

```
import java.sql.*;
public class sqlprogram {
public static void main (String args[ ]) {
Statement stmt;
ResultSet rs;
  try {
    try {
      Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
    }
    catch (Exception e) {
            System.out.println("\n Class Not Found");
    }
    Connection con = DriverManager.getConnection
("url","myLogin","myPasswd");
    Statement stmt = con.createStatement();
    rs = stmt.executeQuery("select * from Events");
    while (rs.next()) {
            System.out.println(rs.getInt("department_id") + ","
            System.out.println(rs.getString("dept_location") + ","
            System.out.println(rs.getString("employee"));
      }//end of while
    }//end of outer try
    catch (SQLException ex) {
      System.out.println("\n SQL Exception "+ exception.getMessage() );
    }
```

```
26.     stmt.close();
27.     con.close();
28.     }//end of main
29. }//end of class
```

Figure 5.2: A Sample of JDBC™ Application

The following describes the steps in detail:

1. Importing the Java.sql.package:

The JDBC™ API is a set of different classes and several interfaces. The name of the package for these classes and interfaces is *java.sql* and is imported at the beginning of the class (Figure 4.2, line 1).

This import statement is necessary for any application that falls back on the JDBC™ API and intends to use this API.

2. Loading and Registering the Driver:

Two different steps must be accomplished. First, the driver must be loaded and second, the driver must be registered to itself. By using the code in Figure 4.2, line 9, an instance of a driver is called and this performs the loading.

The *Class.forName()* method takes the complete package name of the driver as an argument. In order to get the actual name of the driver, the programmer has to refer to the vendor's documentation.

3. Establishing a Connection to a Database:

Next, a connection to the database must be established by using the appropriate *DriverManager.getConnection()* method. The method takes at least two arguments. The first argument is a string that represents the URL of the database, and the second

argument is a set of login properties, like the user's name and the appropriate password. The code in Figure 4.2, line 14 illustrates how this is accomplished.

4. Sending SQL Statements:

Next, a software developer must create an SQL statement object from a user-interface to a database in order to execute the query. The statement to be executed is provided to the appropriate method of the statement object. The code in Figure 4.2, line 15 illustrates the creation of a statement object.

This is one of three different types of statement objects. There are two other types of statements available, the *PreparedStatement* and the *CallableStatement.* Both are subclasses of the Statement class.

5. Execution of a Statement:

This step provides the actual execution of an SQL statement. This method *Statement.executeQuery*() is used to execute a simple query and to receive the result rs. The code in Figure 4.2, line 16 illustrates the execution of a statement.

The method takes an SQL query string as an argument and returns the results of the SQL query as a ResultSet object. The object contains the information data and the methods for retrieving the data.

6. Retrieving the Results:

After the SQL statement execution, the next step is to retrieve the results. They are stored in a *ResultSet* object. In order to retrieve data from the *ResultSet* into a Java variable, one must invoke the *ResultSet* get-method. The get-method converts the data into a Java type. The method takes the column index as an argument and returns the value that is found in the current row of that column. When implementing the get-

method in a loop, one can step through the entire result set. The code in Figure 4.2, line 17-21 illustrates the retrieval of the result by using a loop.

7. Closing the Connection and Statement:

This is the final step of any database client-server application. All open connections can cause severe security problems; therefore, it is recommended to close the connection immediately after the last call. The code in Figure 4.2, lines 26 and 27 illustrates the closing part of the application.

THIS PAGE INTENTIONALLY LEFT BLANK

VI. CONNECTIONS TO THE CLIENTS AND THE INTERNET

A. INTRODUCTION

The expression client-server was first adapted during the 1980s in conjunction with computers connected to the network environment. Models of client-server were accepted by users and developers in the late 1980s and were intended to improve the usability, the flexibility, the interoperability, and the scalability of PC-based computers compared to main-frame computers with time-sharing and centralized computing design. A client is defined as a requester of a particular service, and a server is defined as a provider of the ordered service. Nevertheless, a computer can act as a client and as a server at the same time depending on the software configuration. In the following, there are some common client-server architectures introduced and described that are both popular and often used in network systems. For the prototype, a connection to the clients is established via Java and JDBC™.

B. CLIENT–SERVER MODELS

In this chapter, different types of client-server computing are introduced. Brief descriptions of one-tier, two-tier, and three-tier architectures are provided. In addition, the *Middleware* concept is discussed.

Nowadays, most application programs have three major layers. The first layer is the *presentation layer,* which provides the human/computer interaction-also called the

user interface. This presentation layer receives the input from the keyboard or other devices and handles the output.

The second layer, the *application layer,* deals with the application or business logic that provides the character of the application program. For example, the application logic provides the difference between an order entry system and an inventory control system. Frequently, it is called *business logic* because the program contains the business rules that describe the workflow in an enterprise.

The third layer, the *service layer,* provides general service for the other layers, like file service, print service, communication service, and the service for a database.

The number of tiers in a client-server application is determined by how tightly the three program layers are integrated.

1. One-Tier Architecture

A one–tier application is an application in which the three program layers are tightly connected. In this case, the *presentation layer* has especially detailed knowledge of the database structure. The *application layer* is often interwoven with both the *presentation layer* and the *service layer.* All three of these layers, in addition to the database engine, run on the same computer system.

One–tier applications are easy to design and easy to write. Furthermore, one can create a multi-user one–tier application by running the application on various computers and letting them share the database system. The database can be stored either on one of the computers (peer-to-peer solution) or on a file server. Multi-user one–tier solutions are feasible until the number of users becomes large. The problem here is that the entire database work is performed in each client. All information including indexes and data

records needed to resolve the query to the database must be transmitted over the network. This reduces the performance because for some queries, a significant amount of data has to be examined, in some cases the entire database.

The solution for the above mentioned performance problems with multiuser one-tier programs is two–tier client-server architecture.

2. Two–Tier Architecture

The two-tier client-server architecture is written similarly to a one-tier application, with the exception that the database service, the engine, no longer runs with the client. Instead, it runs on the server computer that contains the database service itself. There are some methods required to communicate between the application logic layer and the database service. Two-tier applications are more complex to write.

In a two-tier system, the client is defined as the first tier and the server is defined as a second tier. In a JDBC™ environment, the database application is treated as a client and the DBMS is treated as a server. The client communicates directly with the server without any help from any other server. The following Figure 6.1 illustrates the two-tier model:

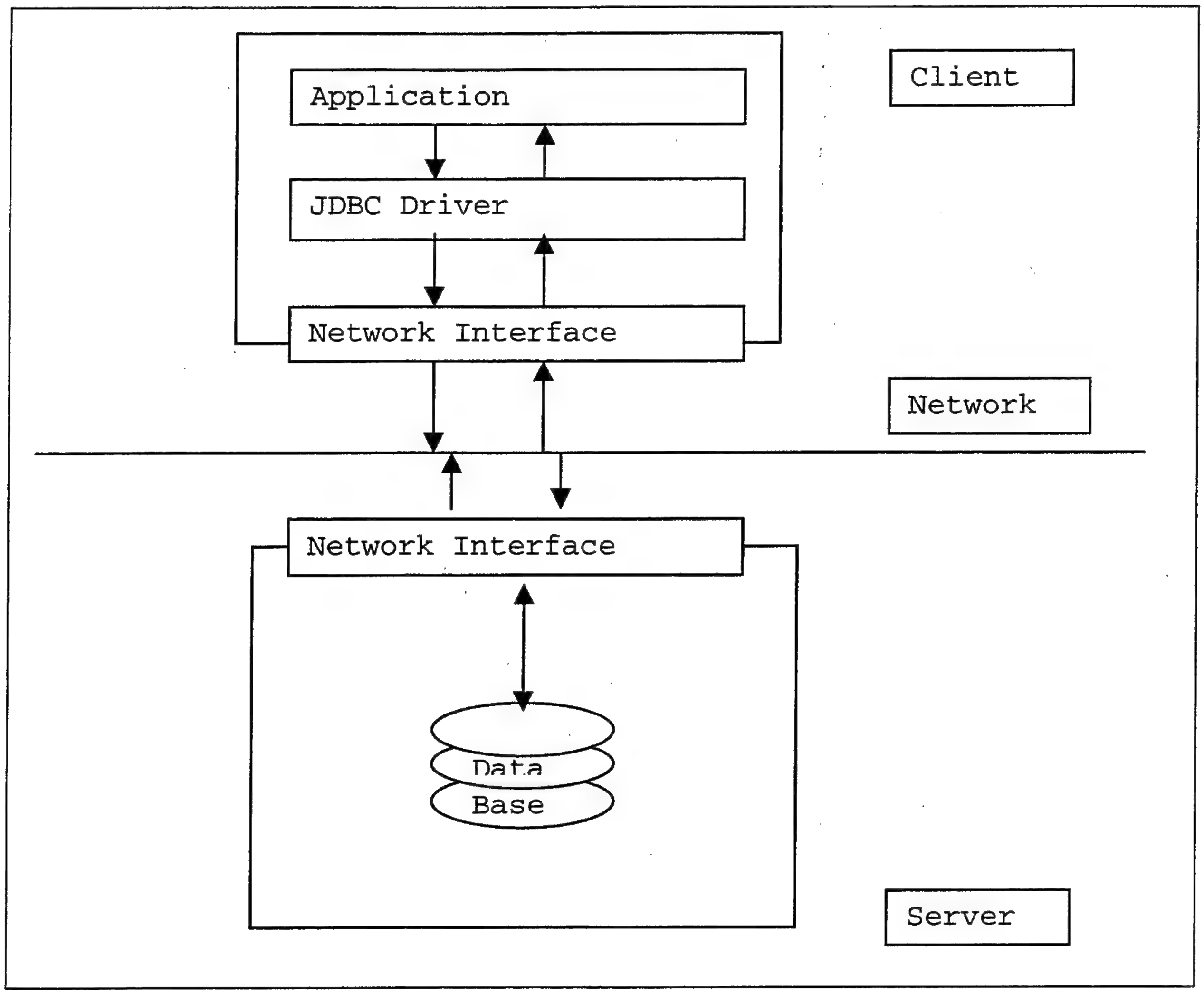


Figure 6.1: Two-Tier Client-Server Model

In a two-tier application, the most common query method is the structured query language (SQL), a query language that encapsulates complex database queries in relatively compact sentences. The SQL statement is sent via the appropriate driver to the database server, which performs the work locally on that computer and returns only the relevant results of the query to the client. The JDBC™ driver is responsible for presenting the query to the database in a manner acceptable to the database.

In a two-tier model only the database services are separated from the application. The presentation and the business logic layers remain very entangled, and both layers continue to have profound and solid knowledge of the database [SIPL98]. While two-tier models offer a great deal of flexibility and simplicity in management, they also have a few disadvantages.

The advantages of a two-tier model are

- A two-tier access model is relatively easy to implement compared to a higher three-tier model.
- The two-tier model keeps the connection between client and server open. Therefore, the overhead associated with establishing a connection is eliminated.
- The two-tier implementation is faster compared to a three-tier model implementation.

The disadvantages of a two-tier model are

- Most of the available drivers require that native libraries be loaded on the client computer.
- Local configurations must be maintained for native code if the driver requires this.
- Applets can only open connections to the server from which they were downloaded.

3. Three-Tier Architecture

The three-tier architecture emerged to overcome the limitations and the disadvantages of the two-tier architecture. In the three-tier architecture, a middle element was added between the user-system-interface-client environment and the database management server environment. This model has the advantage of allowing the user to separate the database server from the Web server. The client requests for the database are directed through a proxy server, which results in a more secure environment for the database. The middle tier can perform queuing, application execution, and database staging.

The three-tier client-server architecture has been shown to improve the performance for groups with a large number of users and also improves flexibility when compared to the two-tier architecture. In this model, the driver translates the requests into a network protocol and then makes a request through the proxy server. The proxy server itself makes database requests on behalf of the client and next passes the results back to the client after they have been processed by the appropriate database system. Flexibility in partitioning can be as simple as moving application code modules onto various computers in a three-tier architecture.

The following Figure 6.2 illustrates the three-tier model:

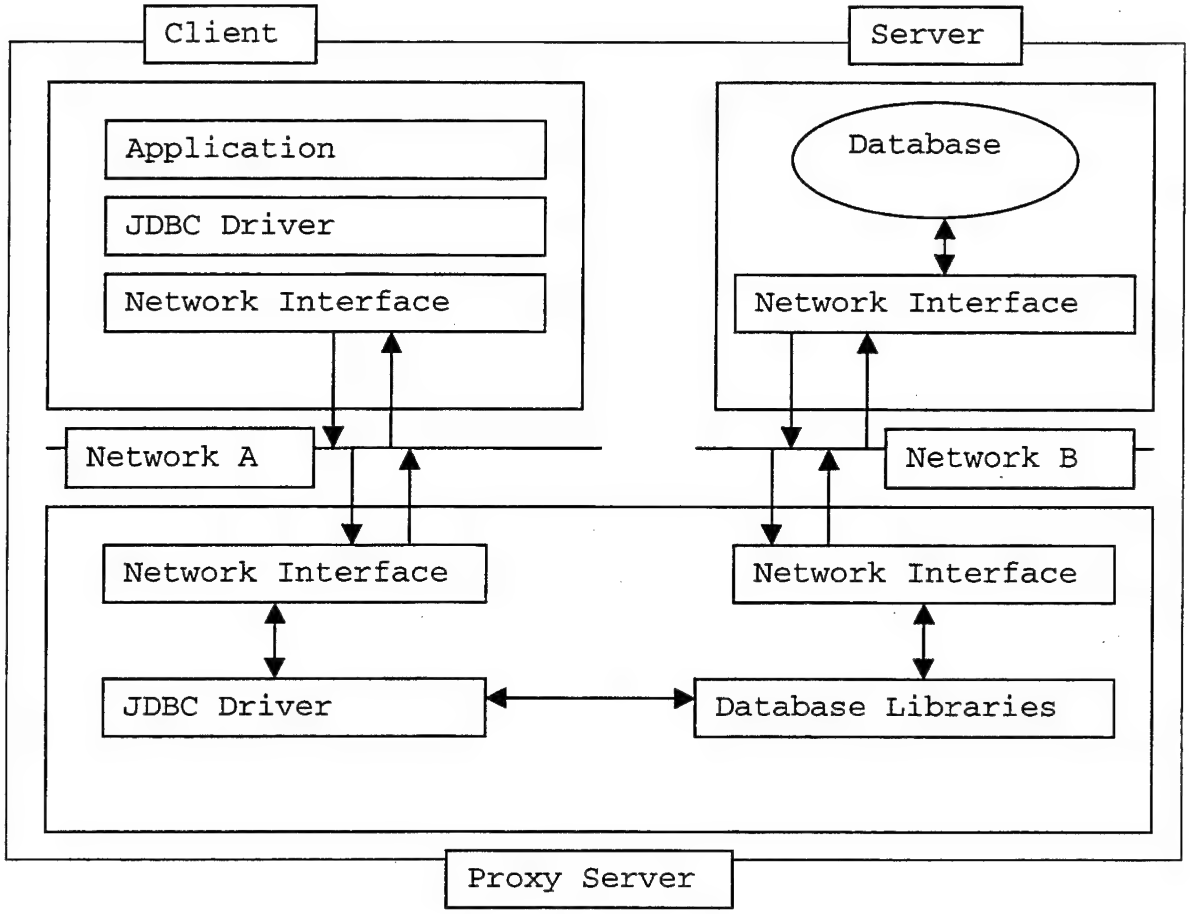


Figure 6.2: Three-Tier Client-Server Model

The following are the advantages of a three-tier model [SIPL98]:

- The clients need not load native libraries locally.
- All drivers can be managed centrally.
- The database system does not have to reside on the same server as the Web server.
- The database server does not have to be directly visible to the Internet.

However the three-tier technology also has some disadvantages:

- The client to server connection does not provide a permanent database connection.
- A separate proxy server may be required.
- An increased network traffic can occur if a separate proxy server is implemented.

In addition to the different client-server design considerations, one must remember how the JDBC™ driver fits into the architectural concept of a client-server application. These drivers can be thought of as a translator between the Java application and the native language of the database. Different driver types are available for different applications. For details about different JDBC™ drivers, different JDBC™ driver categories, and different JDBC™ driver implementations see [HCF97] in Chapter IX.

VII. DISCUSSION OF RMI AND SERVLETS

A. OVERVIEW

This chapter describes the different approaches to using objects in a remote manner. The first section describes how to use Java objects from a different virtual machine by using *Remote-Method-Invocation* (RMI) that is a distributed object technology. RMI resolves the issues surrounding data marshalling by using serialization. It allows the user to read and to write objects in a remote appearance without necessarily being aware that these objects are remote.

The second section describes the Java Servlets that can be thought of a server-side translation of applets. Servlets are a small piece of Java code that is loaded by a Web server, which is used to satisfy client requests. They are persistent, and platform independent. They also incorporate advanced properties like security-related issues, database access, and integration with Java applets.

1. Remote Method Invocation

In a multi-tier JDBC™ application, several difficulties arise in developing a robust and efficient network protocol that can be used to pass data back and forth between the server and the client. These difficulties can be avoided by the use of the RMI framework. Java RMI enables the software developer to create distributed Java-to-Java applications. RMI can be used to transmit objects transparently from one server to another server. It can be programmed to provide service as a proxy server to a database system by setting up a listener process to handle all access requests from the clients.

When access requests are received, RMI can invoke a set of methods that make the request to the database on behalf of the client. The important advantage of using RMI to accomplish this is that RMI is part of Java, and not a vendor-specific implementation.

2. Java Servlets

A different architecture that establishes a connection from the client via a server to a database is the implementation of servlets. Servlets are modules that extend request and response style servers. Servlets are to Web servers what applets are to Web browsers. Unlike applets, servlets do not provide any GUI. Servlets can be embedded on many different server architectures because of the servlet API. The API has no information about the server and assumes nothing about the server's environment or the server-used protocol. Presently, the most widely used server is the *HyperText Transfer Protocol* (HTTP) server. Servlets are an effective replacement for *Common Gateway Interface* (CGI) scripts [HM99] because they provide a way to create documents that are easier to write and faster to run.

The other benefits of using servlets are that they allow collaboration between people of conferences or forwarding a request. A servlet can also handle multiple requests concurrently and can synchronize these requests, and in addition a servlet can advance requests to other servers and servlets in order to balance work load among servers working on the same content. The following sections describe these different architectures in detail.

B. REMOTE METHOD INVOCATION INTERFACE

Remote Method Invocation (RMI) Interface lets the software developer create distributed Java-to-Java applications in which the methods of remote Java objects can be invoked from objects in other Java virtual machines. A Java program can make a call on a remote object once it obtains a reference to the remote object, either by looking up the remote object in the bootstrap-naming-service provided by RMI or by receiving the reference as an argument or a return value. A client can call a remote object in a server, and that server can also be a client of other remote objects. RMI uses Java's object serialization to arrange parameters in the right order and does not truncate types.

For basic communication mechanisms, Java supports sockets that are flexible and sufficient for general communication. But sockets require the client and the server to connect in application level protocols in order to encode and decode messages for exchange. This design is burdensome and can induce errors.

In contrast, *Remote Procedure Calls* (RPC), which do not work with sockets directly, instead provide the developer with an illusion of calling a local procedure. RPCs do not translate well into a distributed object system where the communication between the program level objects residing in different address spaces is needed. At this point, to match the semantics of any object, RMI is needed by any distributed object system. Any invocation on a remote object is supported and controlled by a local object called a *stub*.

The RMI framework has a variety of advantages over traditional RPC systems. RMI is totally focused on Java, with connectivity to existing systems using Java's native interface (JNI).

The following lists the primary benefits of RMI [ARMS98]:

- **RMI is Object Oriented:** RMI can pass objects as arguments and return values. It is possible to sent objects directly over the connection without an extra code for the client. The user can pass complex types as an argument (for example: hashtable objects).
- **RMI is Safe and Secure:** RMI uses the built-in Java security mechanisms, which allow the computer system to be safe when the user downloads executable code. Java handles security via the SecurityManager object, which delivers a judgment on all security sensitive actions, like opening files or establishing a connection to a network. RMI uses this standard Java mechanism and requires that the security manager is installed before any method on the server is invoked. RMI provides a very restrictive security manager type, permitting only connections to the originating host. Also no file access is allowed. This prevents downloaded implementations from reading and writing data from the computer system or connecting to other systems behind an isolated network.
- **RMI is Easy to Write and Easy to Use:** RMI makes it easy to write the Remote Java Server code and the Java Client code. A remote interface is an actual Java interface. A server has approximately three lines of code to declare itself. This simplicity makes it easy to write the server code for large scale distributed applications and allows one to bring up prototypes for testing and evaluation rapidly. Finally, because programs using the RMI framework are easy to write, they are easy to maintain.

- **RMI Write Once and Run Anywhere:** RMI can handle the "write once; run anywhere" approach. Every RMI-based system is totally portable to any Java Virtual Machine. The same applies to an RMI/JDBC™ system.
- **Distributed Garbage Collection:** RMI uses the Java *garbage collection* feature to collect any objects that are no longer referenced in the network. The distributed garbage collection can define server objects as needed, knowing they will be collected and removed when they no longer need to be accessed by any client.
- **RMI allows Parallel Computing:** RMI supports multi-threading. This permits the server to exploit the Java threads for more effective concurrent processing of the client requests.
- **Java Distributed Computing Solution:** RMI is an element of the core Java platform starting with JDK 1.1. It is available on any JDK 1.1 Java Virtual Machine. All RMI systems use the same public protocol; therefore, all Java systems can communicate with each other directly. As a consequence, only a little "protocol-translation-overhead" exists.
- **Connections to Existing Systems:** RMI communicates with other systems via Java's native interface (JNI). JNI supplies a platform independent method for calling routines written in other programming languages. By using both RMI and JNI, the developer can write the code for the client in Java and can still use an existing server implementation. In addition, when the developer works with RMI and JNI in order to connect to an existing server, rewriting any element of the server in Java and still getting the full advantages of Java in the

new code is possible. Also, Java RMI will work together with Internet Inter-ORB Protocol (IIOP) [AL99], the transport protocol that is a part of the Object-Management-Group-Common-Object-Request-Broker-Architecture (CORBA) for distributed computing. A transport protocol is a set of message formats that allows information to be passed across the network from one computer system to another computer system. Hereby, Java RMI supports its own protocol and will support other protocols including IIOP. Finally, RMI communicates with existing relational databases by using JDBC™ without changing existing non-Java source code used by the database.

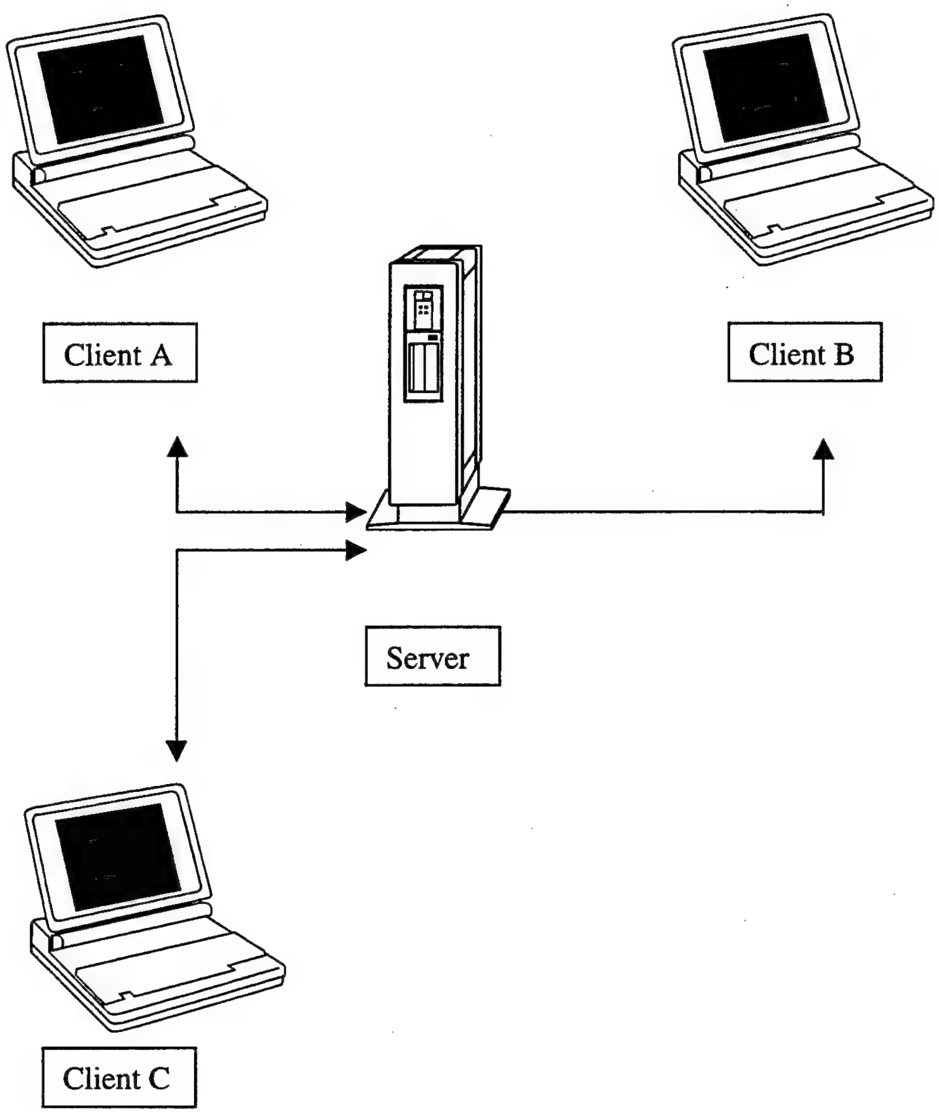


Figure 7.1: The RMI Architecture

1. Introduction to the Java Distributed Model

The Java distributed model is almost the same as the Java object model. Any reference to a remote object can be sent as an argument, or it can be received as a result in any method invocation. This is possible for a local and remote invocation. A remote object can be cast to a remote interface. This is supported by the implementation of the Java syntax for casting. On the other hand, there are some differences between the Java distributed object model and the Java object model. Any clients of remote objects work with remote interfaces only. They never work with the implementation classes of the interfaces. All remote objects are passed by reference instead of creating and copying the actual remote implementation. Finally, the semantics for some methods that are defined by the class *Object* are specialized for remote objects.

The interfaces and the classes responsible for specifying the remote behavior of the RMI framework model are all defined in the *java.rmi* and the *java.rmi.server* packages. The methods in a remote interface are defined as follows. First, each method must declare a *java.rmi.RemoteException* in its throws clause, and second, the remote object passed as an argument or return value must be declared as a remote interface instead of the implementation class.

With Java implementation, parameter passing in RMI is always possible. An argument to or a return value from a remote object can be any Java type, which is serializable. For example, Java primitive types, remote Java objects, and also non-remote Java objects that use the *java.io.Serializable* interface are included.

2. Architecture

a) Introduction

The RMI framework architecture is designed to establish a solid base-structure for distributed object-orientated computing. The architecture is planned and developed to allow for future implementations of server and various computers, so software and hardware can be added in a congruous manner. For example, if a server is exported, a special reference type is defined. Usually, servers are exported as a *UnicastRemoteObject*, which means they are point-to-point unreplicated servers. Other types of servers are defined as *MulticastRemoteObject*, a reference semantic approved for replicated service.

b) Overview

The RMI framework system contains three different layers: (1) the stub and skeleton layer, (2) the remote reference layer, and (3) the transport layer. Each layer is totally independent of the next layer; furthermore, one layer can be replaced by an alternate implementation, without influencing the other layers in the system. The first layer is responsible for the connection of *client-side-stubs*, also called *proxies*, to server side *skeletons* and vice versa. The second layer is responsible for the remote reference behavior, like the invocation to a single object or the invocation to a replicated object that requires communication with several locations, and the third layer is responsible for setting up and managing a connection and is responsible for tracking the object. The

application layer—hosting the client and the server—is implemented on top of the RMI framework system. The communication between the client and server works as follows:

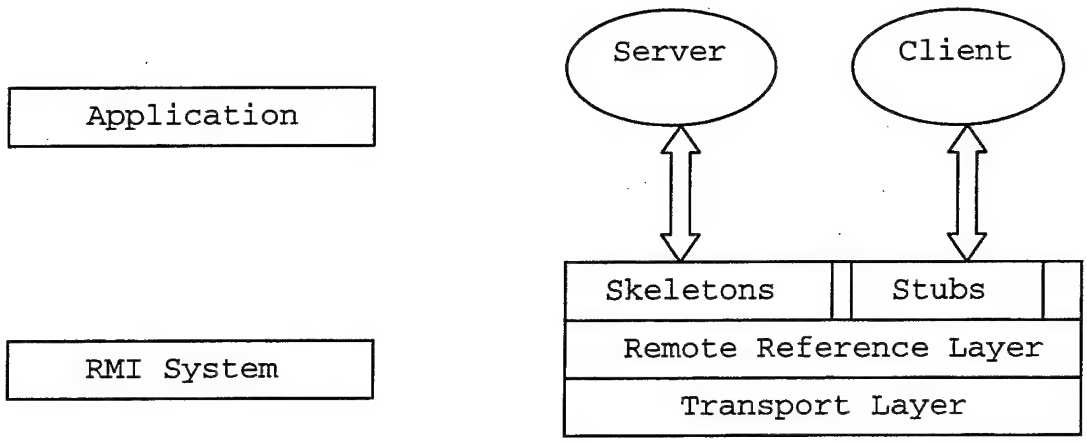


Figure 7.2: Communication between Client and Server

c) The Stub and Skeleton Layer

A *skeleton* is a server-side element, which contains a method that sends off calls to the actual remote object implementation. *Skeletons* are responsible for (1) unmarshalling arguments from the marshal stream, (2) establishing the up-call to the actual remote object implementation, and (3) arranging the return value of the call or an exception back onto the marshal stream in the right order.

A *stub* is the client-side proxy for the remote object. The primary task of the *stub* is to implement all interfaces, which are supported by the remote object implementation. *Stubs* are also responsible for four other important functions: First, they establish calls to the remote object. This is done by calling the remote reference layer.

Second, they are responsible for arranging the arguments onto the marshalling stream. Third, they are responsible for notifying the remote reference layer that the call should be invoked, and finally, they are responsible for notifying the remote reference layer that the call is complete.

The *Skeleton/Stub layer* connects the application layer and the RMI framework system. The *Skeleton/Stub layer* transmits data to the remote reference layer by abstraction of marshal streams. These streams instruct a special mechanism (object serialization) to allow Java objects to be transmitted between address spaces. These objects are usually passed by copy, unless they are remote objects. In this case, they are passed by reference.

Stub and *skeleton classes* are determined at run time and are dynamically loaded when needed.

d) The Remote Reference Layer

Next, the *Remote Reference Layer* is responsible for the lower-level transportation interface. Beside this, the layer is responsible for getting the specific remote reference protocol done. The protocol is independent from the client stubs and the server skeletons. The remote reference layer consists of two components, the client-side component, and the server-side component.

The first component provides specific information about the remote server and also communicates with the server-side component via the transport layer. A special remote reference semantic is used during each method invocation. The second component is responsible for implementing the remote reference semantics to the

skeleton before it is delivered. Lastly, the *Remote Reference Layer* is connected to the transport layer and provides the data for transportation.

e) The Transport Layer

Lastly, the *Transport Layer* of the RMI system sets up, manages, and monitors the connection. It also listens for incoming calls and sets up the connection. Finally, it maintains a table of all remote objects which populate the address space. The remote object reference contains an *endpoint* and an object identifier. After this, a transportation can use the *endpoint* for setting up a connection to the address space in which the remote object resides. On the other side is the server, which receives the object identifier and checks for the target of the remote call.

The transport layer contains four abstractions:

1. The *endpoint* used to determine an address space or a Java Virtual Machine.
2. The *channel* to establish a connection between two address spaces. The channel is responsible for the connection management of the local address space and the remote address space.
3. The *connection* itself for transporting data by performing the input and the output.
4. The *transport* that manages the channels between two addresses spaces. It is also responsible for accepting all calls on incoming connections to the address location, for setting up a connection object for the call, and is responsible for dispatching the calls to the upper layers in the system.

The transport determines what the representation of the endpoint is; hence, several implementations may exist. All the RMI framework transport interfaces are available to a Virtual Machine implementation, but not directly to the application.

f) The Java Garbage Collection

For the distributed system having an automatic feature for deletion of those remote objects which the client no longer requires is desirable. RMI uses a reference counting garbage collection procedure and controls and counts the live references within the Java Virtual Machine. Whenever a live reference occupies the system, the reference counter will increment by one and when live references are found unreferenced in the Virtual Machine, they are withdrawn, the counter is decremented, and the server is notified. As long as a local reference to a remote object exists, the object cannot be garbage collected and can be returned to the clients.

However, there is a disadvantage to garbage collection. The premature collection of objects can happen because the transport layer may believe that the client has crashed. Hence, remote references are unable to guarantee referential integrity, which means in some cases, remote references may not point to an existing object. This erroneous reference will create a *remote exception* and has to be handled in an application.

g) The Dynamic Class Loading

The *remote procedure call* (RPC) and the RMI framework system perform class loading differently. The RPC system recognizes the procedure by generating the

client-stub code and linking the code into a client before an RPC can occur. This code can be linked either statically into the client, or it can link the code at run time by using the appropriate dynamic linking with libraries. These libraries are available either locally or are available over the network file system.

On the other hand, RMI does generalize the above-mentioned technique. RMI uses a special mechanism that loads the required classes at run time. The following classes are required:

- The classes for the remote objects and the interfaces
- The stub and skeleton classes which serve as proxies for the remote objects
- All other classes that are used directly by an RMI application, like parameters to RMI, or return values from RMI

Furthermore, the dynamic class loading instructs two other mechanisms:

- The *object serialization* system to transmit all classes over the network.
- The *security manager* that checks all classes which are loaded into the system

In order to create transient applications, having the technical possibility to store and retrieve Java objects for future use is important. The typical way to handle the objects is by representing the state of the objects in a *serialized* form. This particular form must be able to reconstruct the object or objects. The task of the serializeable form is to identify and to verify Java classes from which the object's contents were saved. Furthermore, restoring the contents to a new instance must be feasible. The stream itself

contains sufficient information to restore the fields in the stream to a compatible version of the class. All stored objects refer to other objects. In order to maintain the relationship between objects, all objects must be stored and retrieved at the same time. As a result, whenever an object is stored, all objects having a relationship to that particular object must be stored as well. The goals for serializing Java objects are listed as follows:

- Have a simple extensible mechanism
- Maintain the Java object type and safety properties in the serialized form
- Allow the object to define its external format
- Require per class implementation only for customization
- Be extensible to support persistence of Java objects
- Be extensible to support the marshalling and unmarshalling, as needed for the remote objects

All classes loaded from the local classpath are considered trustworthy and are not restricted in their behavior by the *security manager*. The security manager must check all classes imported over the network. The Java program starts the security manager early in order to check and to manage all following program steps. The security manager also provides insurance so that all loaded classes adhere to the standard safety rules that Java provides. Good examples are applets that are loaded from a trustworthy server. These applets do not try to use or to manipulate any sensitive methods. This is the primary advantage of applets. They are always subject to restrictions provided by the

AppletSecurity class. The security manager is responsible for providing classes only from the applet server.

3. Client Interfaces

The RMI framework provides various types of interfaces when writing an application or an applet. First, in the Java programming language, a remote object is an instance of a class implementing a *remote interface*. The remote interface is used to declare every method the programmer would like to call remotely. These remote interfaces have the following characteristics:

- The remote interface must be defined as *public*; otherwise the client will receive an error when the program attempts to load an implementation of the interface.
- The remote interface extends the *java.rmi.Remote* interface.
- All methods must declare *java.rmi.RemoteException* in the *throws* extension. This declaration has to be done in addition to any other application specific exception.
- The data type of any remote object, which is passed as an argument, or a return type must be declared as a remote interface type, not as an implementation class.

  Example:

  ```
  Package java.rmi;
  Public interface Remote {}
  ```

A second type of class provided by RMI is the *RemoteException class*, a subclass of *java.rmi.RemoteException*. This allows an interface to deal with several types of remote exceptions and to distinguish local exceptions. It can be constructed with a *throwable* (nested exception). Usually, the nested *(Exception ex)* is the underlying I/O exception that takes place during an RMI call.

Example: `public RemoteException(String name, Throwable ex);`

Lastly, the *Naming class* permits remote objects to be retrieved and defined by using the uniform resource locator (URL) syntax. The URL itself contains the protocol, the host, the port, and the name fields. The protocol is specified as an RMI.

Example: `rmi://java.sun.com:2001/root`

4. Security

In order to achieve and maintain security, a *SecurityManager* object has to be established for security check methods. Any security breach detected by the security checks induces a *SecurityException* error and an associated warning directive to inform the user.

The security manager must be initiated as the first action of a Java program; hence, it can control the following steps of the program. The security manager ensures that loaded classes adhere to the standard Java safety features. For example, it ensures that classes are loaded from a trusted source, and also ensures that they do not attempt to access sensitive functions. All applications must either define their own security manager or use the restrictive *RMISecurityManager*. When an application defines its own security

manager, classes are loaded by using the default *Class.forName* mechanism. Therefore, a server might define its own policies by using the security manager and class loader, and the RMI framework system thereby operates in those policies [WANG98].

C. COMPARISON BETWEEN MULTI-TIER AND RMI POLICY

This section describes the difference between the multi-tier policy and the RMI policy and why the RMI policy is preferred.

1. Similarities between Multi-Tier and RMI Architecture

The procedure for a dynamically configurable information system starts as follows: A client machine displays a GUI to the user who works on the report away from the server machine. All clients communicate with the server by using the RMI framework. The server stores the reports in a database using JDBC™, which is the Java specific relational database package. Up to this point, the procedure resembles a multi-tier system, as described in the previous section.

2. Difference between Multi-Tier and RMI Architecture

The important difference between a multi-tier system and a system using RMI is the ability of RMI to download behavior. Behavior is defined as class implementation. RMI knows how to move behavior from a client to a server and how the client retrieves information from the server. For example, one can define a particular interface for a report (sign-out sheet for Naval Airwing) that represents a special policy. This report can easily be downloaded by the client from the server. Whenever the policy changes, the application on the server starts returning the different implementation of that policy.

Therefore, the constraints will be checked on the client-side, which provides a faster feedback to the user and far less load on the server. Also, an installation of updated software is not required. The benefit of this procedure is that changing policies requires a software developer to write new software only once and then implement the software on the server. This provides a maximum of flexibility and saves time. The intention of this policy is always dynamic and has the following advantages compared to any static approach [JAVAS99]:

- Client machines do not have to be halted when software is updated while the computer system is running.
- The procedure allows dynamic constraints because not only object implementations are passed between clients and server, but also data.
- All users are instantly informed about any errors.

D. JAVA SERVLETS

1. Introduction to the Servlets

The Internet has recently spawned the invention of several new technologies in client-server computing. One of those new developments was the computer language Java. This also serves as a complete client-server solution where applications are automatically downloaded to the client machine and are executed. Much effort has been concentrated on the client-side development of applets and GUI components. Applets on the client side are an important element of the client-server architecture, yet the server side of the architecture is equally important. Servlets can be considered a server-side

applet. They are loaded and executed by a *Web server* in the same way applets are loaded and executed by a *Web browser*.

2. Architecture

Servlets are Web elements that create dynamic content. They are platform independent Java classes compiled to an architecture-neutral byte-code that can be loaded dynamically. Java servlets are controlled and managed by a servlet container. The container is either installed on the Web server or set up as an add-on element to a Web server via the server's API. The container, in alliance with the Web server, furnishes the network service. Furthermore, the container controls the servlets through their lifecycle. The servlet's lifecycle characterizes how a servlet is loaded and initialized, how it obtains queries and responds to queries, and how it is withdrawn from service. Servlet containers have to support HTTP as a protocol for queries and replies. A client program which could be an application, capable of creating a connection across a network can access a Web server and can make an HTTP request. This request is processed by the container running inside the host Web server. Finally, this server calls the servlet. Java servlets are a mechanism for elongating the functionality of servers. The previously mentioned Java characteristic "write once, run anywhere" can be realized on the server as well. Servlets can extend the Web server's functionality in the same manner CGI scripts do. Yet, servlets are clearly much less resource intensive than CGI scripts. In the following section, detailed information about CGI scripts is provided.

3. Comparison between Servlets and CGI

In their basic form, servlets are a powerful replacement for CGI scripts. CGI scripts are typically written in Perl and are usually tied to a particular server platform. Servlets have some benefits over CGI scripts [HM99]:

- Servlets are persistent. They are loaded only once by the Web server and can maintain services, like a database connection, between several queries.
- Servlets provide a better performance since they have to be loaded just once.
- Servlets are platform independent because they are written in Java language.
- Servlets are extensible because they are written in Java, and Java is a robust and object-orientated language, easily extended and adopted to the application.
- Servlets are secure, for the only way to invoke them is through the Web server. This point establishes a high level of security.

4. Information Flow within Servlets

A servlet has a lifecycle that characterizes how it is loaded and initialized, how it obtains queries and responds to them, and how it is withdrawn from service. Every Java servlet has to implement the *javax.servlet.Servlet* interface in order to be run in the servlet engine. This servlet engine is a customized extension to the Web server. The servlet engine creates and loads the servlet. This procedure happens when the engine starts or when the engine needs the servlet in order to answer a query. The servlet engine can load the servlet from either the remote file system, the local file system, or from any file on the network. Next, the servlet engine has to initialize the servlet. At this time, the servlet can read persistent data, can initialize the JDBC™ database connection, or can establish a reference to other resources.

After the servlet initialization, the servlet is ready to receive any queries from the client. Every client request is represented by a servlet request object. The response that the servlet sends back to the client is represented by a client response object. When the client makes a request, the engine passes the client-request-object and the servlet-response-object to the servlet. They are passed as parameters to the service method defined by the service interface.

These servlets can accept a query from the client machine, which processes the request and returns the result. The following list describes the basic information flow:

1. The client machine performs a request via HTTP.
2. The Web server receives the request and provides it to the servlet.
3. The servlet receives the HTTP query and performs some processing.
4. The servlet responses back to the Web server.
5. The Web server forwards the response back to the client.

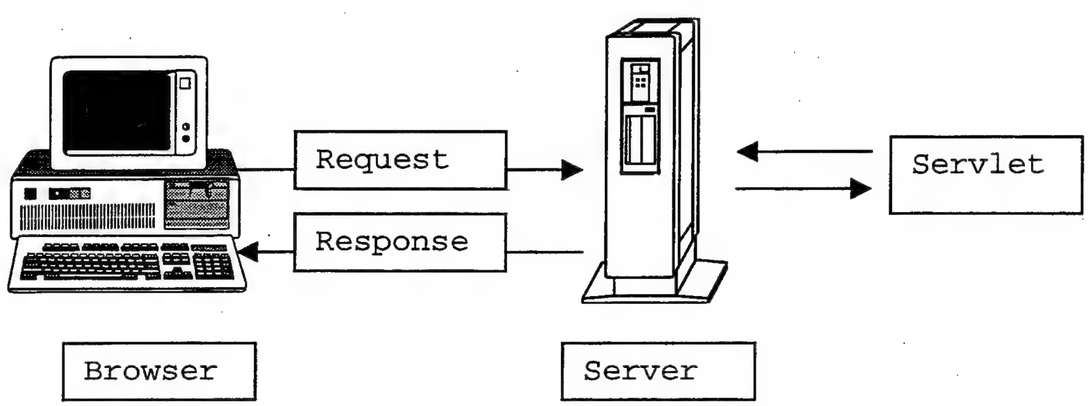


Figure 7.3: Basic Servlet HTTP-Information Flow

One of the important advantages of servlets compared to other systems is that security issues do not apply since servlets are executed on a server. The Web server does not communicate directly with the servlet; instead the servlet is loaded and executed by the Web server. Additionally, if the server is implemented behind a firewall, the servlet is consequently secure as well.

As a result, the basic requirement for running servlets is a Web server that provides and supports the servlet API. A servlet is a Java object that corresponds to a specific interface. These interfaces are defined by the Java server architecture. Servlets are loaded and invoked by services, and a service can use multiple servlets. With servlets, extending the functionality of a server is done easily in two ways. This is feasible either with internal servlets--provided by the server--or it is feasible with user-written servlets that operate as add-ons.

THIS PAGE INTENTIONALLY LEFT BLANK

VIII. SYSTEM DEVELOPMENT AND IMPLEMENTATION

A. OVERVIEW

In the previous chapters, especially chapter VII., different design approaches were discussed, cataloging, as well, the advantages and disadvantages of each design issue. After carefully evaluating and considering each approach, the decision for RMI as an appropriate design for implementing and creating a prototype was made without doubt. The RMI framework protocol provides the best solution for the previously mentioned requirements and the current circumstances at the Naval Airwings. The RMI framework protocol enables any Java object to communicate with other Java objects across a network or the Internet, and it provides nearly the same performance as CORBA or the Microsoft Distributed Computing Environment (DCE). Let us review some of the primary advantages of RMI as detailed in Chapter IV and Chapter VII, pages 34 to 37:

- RMI is Object Oriented
- RMI is safe and Secure
- RMI is Easy to Write and Easy to Use
- RMI Write Once and Run Anywhere
- RMI posses Distributed Garbage Collection
- RMI allows Parallel Computing
- RMI provides Java Distributed Computing Solution
- RMI allows Connections to Existing Systems

Besides these significant benefits, RMI also makes use of the common advantages of the Java language. Moreover, a primary advantage of developing the RMI-based software applications is that learning the CORBA's Interface Definition Language (IDL) is not necessary for the software developer. Hence, the IDL need not be considered.

The RMI framework limits the software development to a pure Java-to-Java application, but includes the benefit of the Java security mechanism when one downloads software over the Internet. Finally, another important financial advantage of RMI is that the software is free of any charge, for it is included as a standard component in the Java Development Kit (JDK).

This chapter describes the database that contains details about the sign-out sheet and information regarding the design of the RMI framework implementation and its operation. The chapter also examines the stages and the decisions made during the system development. This chapter further considers the selection of the MS-Access database product and the intentions for that particular implementation. Following this, the more comprehensive part of the client-server implementation is presented.

B. DATABASE MS-ACCESS

The administrators in the Naval Airwing are still using the "paper and pen" method to record and administer the sign-out sheets. To be sure, a database stored on a computer system is much more flexible and more reliable than a simple paper list.

According to the personnel of "Naval Airwing 2," some COTS products for database implementation are currently available. The database system MS-Access is obtainable and also suitable for managing the data elements and records of the sign-out

sheet. Further, at the Naval Airstations 2 and 3 "Graf Zeppelin," no administrators nor software developers are present. Consequently, some personnel with only limited knowledge of software installation and software alteration has to service the computer program and the computer system.

MS-Access is a powerful and robust 32-bit relational database management system (RDBMS) for creating desktop and client-server applications that run under Windows 95/98 and Windows NT 4.0. The MS-Access database is relatively easy to use with comprehensive wizards designed for inexperienced database users. The database program allows the user:

- To store almost limitless amounts of information
- To organize the information clearly
- To retrieve results based on SQL selection criteria specified by the user

MS-Access is specifically designed for creating multi-user applications where database files are shared on networks; also, it incorporates a sophisticated security system to prevent unauthorized persons from viewing or altering the databases. In addition, MS-Access supports a database structure capable of combining all related data tables and their indexes, forms, and reports [MICR97].

In order to create the database labeled, *MFG 3,* the sign-out sheet presently used at the Naval Airwings was adopted. This form contains all the information regarding flying times and flying events that the crewmember must fill out after returning from their mission.

The database is divided into several tables in order to increase the performance during the query and updating process. Because this was a completely new technology,

there were no available specifications for the database design from any higher command. Hence, the database and the relational tables were designed solely according to the needs of the Naval Airwing and its personnel. As a guideline, I adopted the existing sign-out sheet and considered the observations and recommendations of aircrews and administrators. Beyond this, I created databases for various other administrative tasks in a squadron and then built databases for administering bombing results of aircrews, leave and holidays for the squadron personnel, and academic training for the personnel. Using separate databases increased security options because different user groups could be created, allowing the administrator to instantiate the groups into smaller and more manageable units. This is also supported by the Windows NT operating system.

C. IMPLEMENTATION OF THE RMI NETWORK PROTOCOL

This section will introduce the various classes and methods needed to create the user interface and the underlying program for accessing the database and establishing the database-server-application and the remote connection. To begin, the database-access-component for client use had to be adapted. Therefore, creating a framework for RMI was necessary. The framework consisted of several components. All components are discussed in detail below.

Initially, to create a software application, first an analytic layout has to be made, then permanently updated, and thereafter followed. In general, a description of the problem and the requirements are needed, as well as a description of how the software application can solve these problems and meet the requirements. This procedure is best done by using the Object-Oriented Analysis and Design techniques. So in summary, the

important point of the above mentioned design technique is to consider a problem and to select the appropriate solution from the viewpoint of objects, as demonstrated in the following Figure 8.1. For more detailed information about development of software products and their methodology, the reader should refer to [BROO95]. The three basic steps for any design and implementation are as follows [LARM97]:

1. The analysis phase, where the evaluation of the problem takes place
2. The design phase, where the logical solution is developed
3. The construction phase, where the actual coding is performed

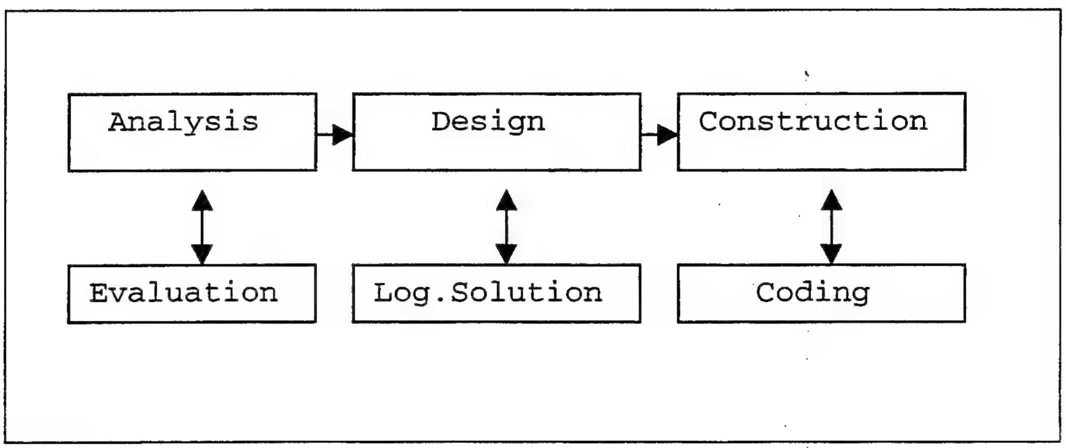


Figure 8.1: Development Steps

As the result of the problem analysis, seven classes with specific responsibilities were identified and were defined for coding and subsequence implementation, as shown in Figure 8.2 on page 62.

Classes	Descriptions
StartupDialog	Provides initial dialog with the user. Queries for name, password and database.
UserApp	Instantiates the RMI User-Application
UserFrame	Provides the connection to the DB and the frame for the application. Also creates the GUI for the User Dialog.
UserProvider	Implements the information provider. DataServerApp is checked for service on the host specified by the HostName properties.
UserResolver	Implements the information resolver.
DataServerApp	Instantiates the RMI server.
DataServerFrame	Displays the frame and GUI for the DataServerApp.
DataServerApp_Skel	Compiler generated Skeleton class
DataServerApp_Stub	Compiler generated Stub class
CrewmemberApi	Defines the methods to use remotely.
ResourceBoundle	Administers messages as key/value pairs and provides appropriate information.

Figure 8.2: Class Listing for Prototype

1. The User Application Class

The first component that had to be developed was the user application, along with the user frame. The user-application-class consists of the *UserApp* method that constructs the application, instantiates the interface and calls the *UserFrame* method. Subsequently, the *pack method* is called. This method initiates the general layout management of the dialog and also sets the initial size appropriately. Next, the method

validates the frame with the preset size, and if validated, *UserApp* makes the frame visible. Also, the main method resides in this *UserApp* method.

The *UserFrame* class is responsible for constructing the GUI front-end to the *UserApp* method. It initializes all the components within a *jbInit* method and calls the database by using the appropriate *setConnection* method. The *setConnection* method includes the jdbc:odbc:driver to the database, the database name, the user name, and the hidden password, if selected.

2. The User Provider and User Resolver Class

After the implementation of the *UserApp* method and its frame, I created a *UserProvider* class that looks up the *DataServerApp* service on the host machine that is specified by the HostName characteristic and properties. The method *provideData* actually fetches the data and stores the contents into the dataSet for processing. Finally, the *getHostName* method is used to locate the *DataServerApp* service by using the RMI. The default call for the host is the local machine.

Next, the *UserResolver* class is created and implemented. First, the *DataServerApp* method is called to look up the appropriate service. The *resolveData* method deduces the changes of the data and stores them in the *DataSet* instance. In the following, the remote method call *resolveCrewmemberChanges* is instantiated and is accomplished. In the following program step, the program checks for any resolution errors and handles these errors accordingly. The program loops through all noted errors and resets a status bit to "not resolved" for each row that initiated an error. Then the program starts again, halts at the first reported error, and lets the provider decline that

error. Finally, the *getHostName* method is used to locate the *DataServerApp* service by using the RMI. The default call for the host is the local machine.

3. The Data-Server Application Class

The data-server application consists of a variety of several classes. First, the *DataServerApp* class acting as an RMI server was created. The class contains a main method in which the *DataServerApp* is instantiated and a *bindToNamingService* is called. In this method, the RMI requires the actual name of the host. Whenever another service has already been registered to the RMI with the name *DataServerApp,* an error is thrown and handled by the program appropriately.

4. The CrewMember Class

The *CrewMember* class contains and defines all the methods I intend to remote. The *CrewMember* class will define all the methods available to the clients. With the RMI, remote objects are not actually downloaded from the server to the client, but rather only references, called handles, are downloaded to the client machine. If the objects are executed, they perform their tasks on the server instead of on the client machine. Consequently, all results and return values from the server are first serialized and then returned to the client.

5. The Resource Bundle Class

At this point, all necessary messages for the various classes and methods in a single resource class were defined. Java provides a special class called *Resource Bundle* for administering messages as key/value pairs. At run-time, the program uses the *get.Bundle* method to retrieve all necessary data and predefined information to load the appropriate *ResourceBundle* class. The *ResourceBundle* class contains all the annotations

the different classes use, each associated with a key, which serves as the message name. The primary advantage of using this method is all information is gathered in one single class, and if one annotation has to be changed, it can be done in this class and subsequently passed to all the different locations in the code [FLAN97].

D. SYSTEM REQUIREMENTS AND COSTS

Before the application program is deployed on any portable computer system, the user should ensure that the system (notebook) meets the following minimum requirements for the installation and the subsequent processing of the program. The system was designed and developed on an Intel computer system running a Pentium CPU with 400 MHz and 128 MB of memory (RAM) which was sufficient for coding and testing. Further tests on a notebook with 300 MHz and 64 MB RAM were performed but showed a reduction of performance in response time. Receiving the results of the queries and displaying them on the screen was rather slow. When sending several queries from different locations across the Internet to the server, a lack in response time is expected and has to be evaluated after a comprehensive field-testing.

A server with the approriate hardware performance and the software connecting the server to the Internet, as well as the software for the database, are available at sites where the system will be deployed. Hence, the required cost for procurement of hardware is minimized, yet labor costs and additional costs for the Java compiler are to be expected.

E. SYSTEM OPERATION

After installing the application on the server and on the client computer, the system must be configured by the user. The user must run the RMI registry before the data server program can be started. The RMI registry is a simple server-side name server that allows remote clients to get a reference to a remote object. The name server is typically used to locate the first remote object the RMI client has to talk to. The next steps include starting the data server program and the user application program. After starting the user application program, a dialog between the user and the system is initiated and a *StartUp* window appears. The user must authenticate himself or herself and must select the required database. Next the user has access to the database and can change data values and send them to the server at the home base for final storage. A screenshot of the *StartUp* window is shown in the following Figure 8.3 on page 67.

The shown GUIs are limited to the main functions for demonstration purposes only. The main goal of the thesis is realizing the connection to the home base and transmitting the data from a remote site; although, sophisticated GUIs are not the focus of this thesis, they should be improved following intensive field testing and further feedback from the user. An improvement of the GUI can be considered as one of the items for future work.

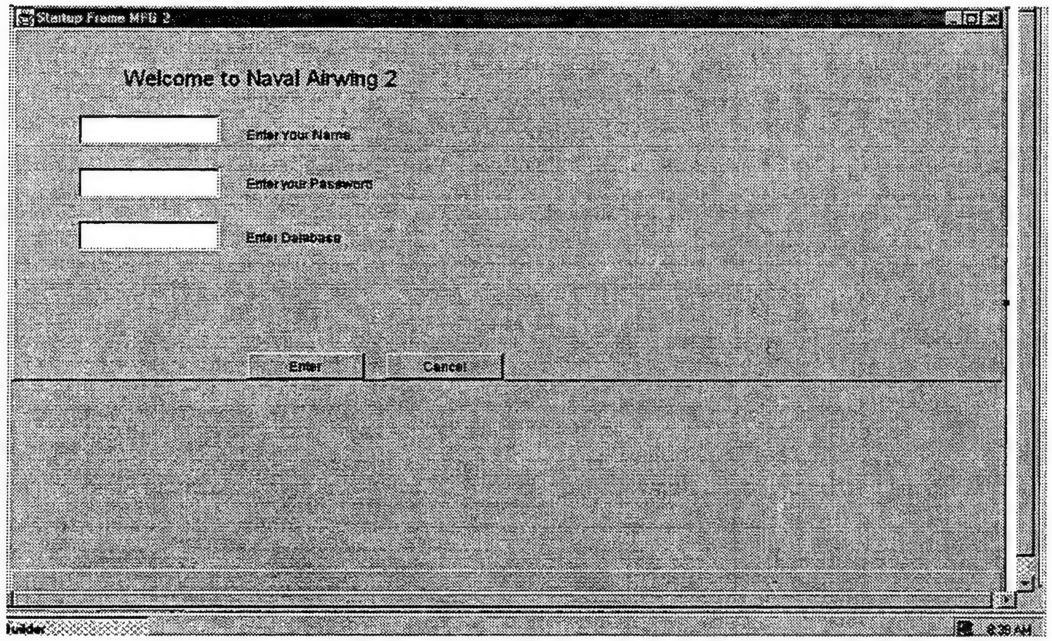
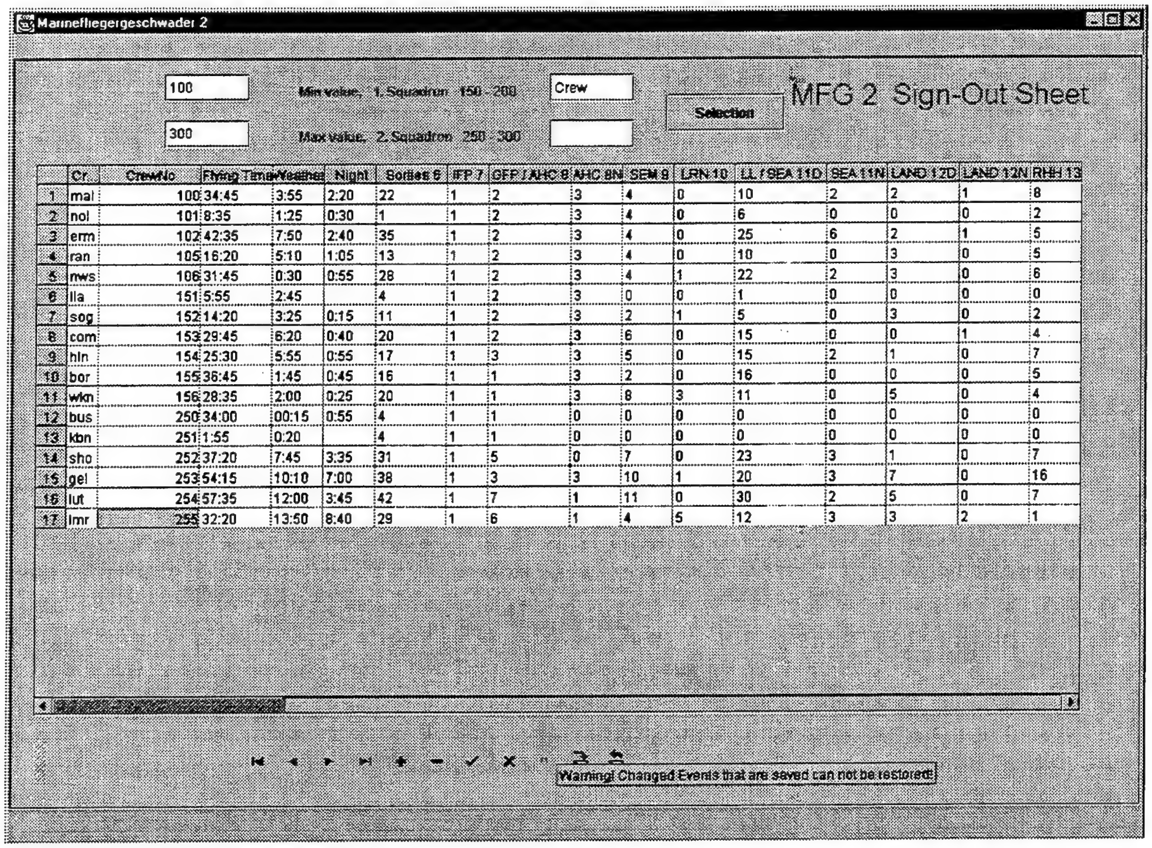


Figure 8.3: StartUp Dialog

When the user selects the MFG 3 database, the system downloads the appropriate data for the sign-out sheet. The system provides information according to the selected CrewNumber. Information can be limited by providing the appropriate range in Crewnumbers, for example, for the first for squadron selection CrewNo 150-200, or by selecting the three-digit name (lut) abbreviation. The later case just shows the single user data for updating. A screenshot of the sign-out sheet with complete data is shown in Figure 8.4 on the following page.



	Cr.	CrewNo	Flying Time	Weather	Night	Sorties 6	IFP 7	GFP / AHC 8	AHC 8N	SEM 9	LRN 10	LL / SEA 11D	SEA 11N	LAND 12D	LAND 12N	RHH 13
1	mal	100	34:45	3:55	2:20	22	1	2	3	4	0	10	2	2	1	8
2	nol	101	8:35	1:25	0:30	1	1	2	3	4	0	6	0	0	0	2
3	erm	102	42:35	7:50	2:40	35	1	2	3	4	0	25	6	2	1	5
4	ran	105	16:20	5:10	1:05	13	1	2	3	4	0	10	0	3	0	5
5	nws	106	31:45	0:30	0:55	28	1	2	3	4	1	22	2	3	0	6
6	lla	151	5:55	2:45		4	1	2	3	0	0	1	0	0	0	0
7	sog	152	14:20	3:25	0:15	11	1	2	3	2	1	5	0	3	0	2
8	com	153	29:45	6:20	0:40	20	1	2	3	6	0	15	0	0	1	4
9	hln	154	25:30	5:55	0:55	17	1	3	3	5	0	15	2	1	0	7
10	bor	155	36:45	1:45	0:45	16	1	1	3	2	0	16	0	0	0	5
11	wkn	156	28:35	2:00	0:25	20	1	1	3	8	3	11	0	5	0	4
12	bus	250	34:00	00:15	0:55	4	1	1	0	0	0	0	0	0	0	0
13	kbn	251	1:55	0:20		4	1	1	0	0	0	0	0	0	0	0
14	sho	252	37:20	7:45	3:35	31	1	5	0	7	0	23	3	1	0	7
15	gel	253	54:15	10:10	7:00	38	1	3	3	10	1	20	3	7	0	16
16	lut	254	57:35	12:00	3:45	42	1	7	1	11	0	30	2	5	0	7
17	lmr	255	32:20	13:50	8:40	29	1	6	1	4	5	12	3	3	2	1

Figure 8.4: Screenshot of the Sign-Out Sheet

A screenshot of the sign-out sheet with filtered data, i. e. the first Squadron only, is shown in Figure 8.5.

The home-base administrator can navigate through the sign-out sheets according to the needs for any changes. The administrator can also insert rows for creating new crewmembers, can delete rows when necessary, and can even filter crewmembers for special events or flying times.

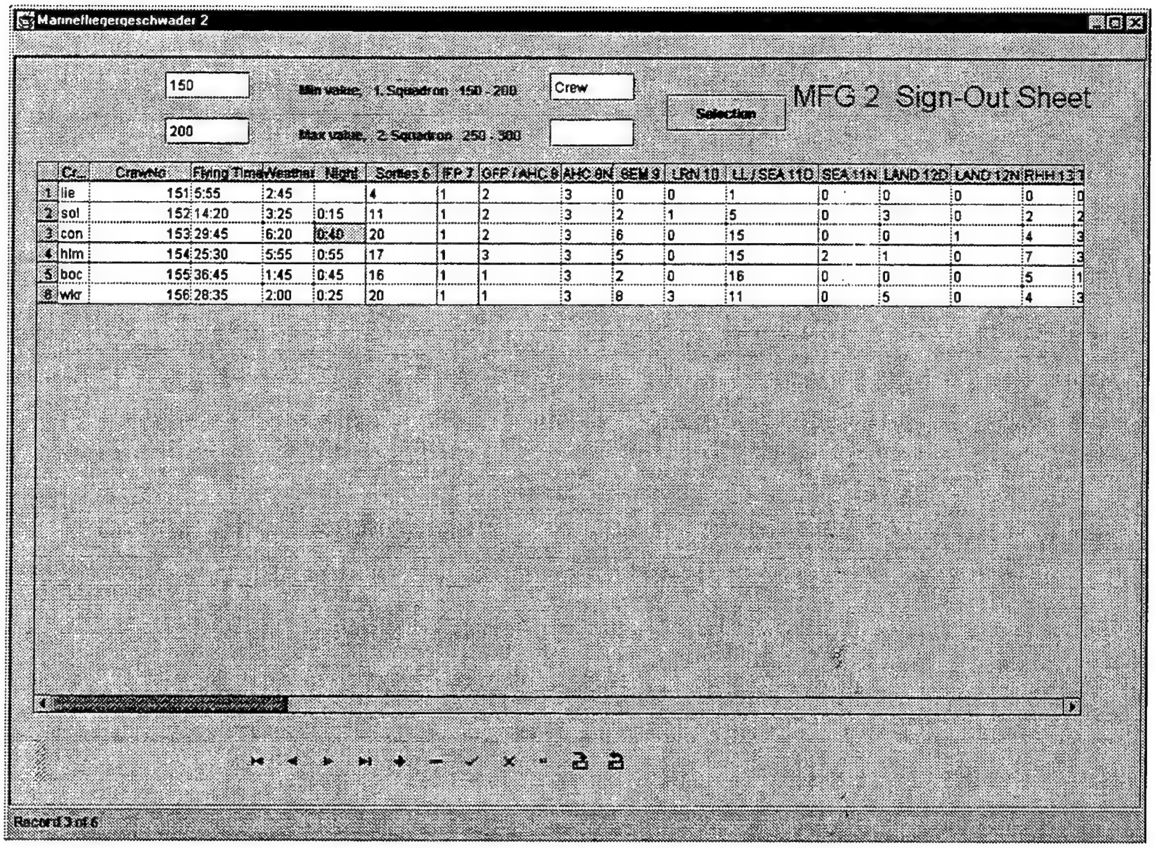


Figure 8.5: Screenshot of the Sign-Out Sheet (Single Squadron)

Initially, all changed information is saved and stored only on the local machine. The administrators, like the deployed crewmembers, have the opportunity to review their changes before they return the modified data to the server on base. Subsequentially, all data is stored in the database and can be used for further evaluation by commanding personnel.

THIS PAGE INTENTIONALLY LEFT BLANK

IX. CONCLUSION AND FUTURE RESEARCH

A. CONCLUSION

With the beginning of increased flying activity in foreign NATO countries and the continuous rise of unit deployment to allied airfields, the personnel at the Naval airbases require an ever-increasing demand for information about their personnel and aircraft. Presently, the Naval airwings, as well as other units are undergoing considerable cuts in financial resources, personnel, and also materials. This permanently changing nature of the airwings has forced the station commander and the staff to handle an increased amount of information with far less personnel. Likewise, providing this essential information is ever more important. Therefore, adapting and relying on newer distributed client-server technologies for an immediate information update and exchange is reasonable. Fortunately, there are numerous designs and architectures available, from which a software developer can choose, depending on the requirements of an organization.

In this thesis, I focused on the design and the implementation of the component-based client-server solution that uses the Internet for the communication. This client-server protocol grants a strong and continuous platform for object-oriented-distributed-computing. By using the RMI framework, one can expand the capabilities of the Java language and obtain all the benefits; namely, low maintenance costs, and a secure environment. RMI provides an independent platform to the user, capable of expanding Java into any part of the system in an incremental manner so that the user can add servers

and clients whenever required. These characteristics were a major factor for choosing the RMI framework technology for the development and the implementation. Units like the Naval Airwing can expand or abridge the system according to their needs and can maintain the system with personnel lacking comprehensive knowledge of computer science. Also the costs are relatively low, as the system was designed and implemented with available COTS products. Consequently, the thesis has proven that connecting a deployed unit to the homebase across the Internet by using the RMI protocol for information retrieval and information update is not only possible but also highly effective and economical.

B. SECURITY

Most of the computer systems connected to the Internet are vulnerable to various attacks. They can compromise the confidentiality, the integrity, and the availability of information. As a first step toward eliminating such dangers, solution technologies include the implementation of the Java-Security-Manager on the software side and the implementation of access controls, strong authentication, and a firewall on the hardware side. As a second step, advanced technologies include data-encryption, the reference monitor concept, and secure protocols.

Before one starts to implement any security tasks, defining the security requirements for the system is essential. For any military data-transmission, the security class should be at least "Confidential"; therefore, a secure channel between the client and the server should be established. The RMI framework provides a *socket factory* capable of establishing sockets of any type, including encrypted sockets. The

java.rmi.server.RMISocketFactory class installs a default socket factory, a valuable resource provider for the client and the server sockets. This factory provides a firewall-tunneling mechanism, operating as follows [Java99a]:

- The client socket automatically attempts an HTTP connection to the host that cannot be connected by a direct socket.
- The server socket automatically checks if the new connection is an HTTP POST request. If it is an HTTP request, the server returns a socket that exposes only the body of the request and produces the output as an HTTP response.

With the introduction of the JDK 1.2, software developers can specify requirements on the services provided for the server's sockets. These requirements have to be described by the software developer and can also include data-encryption. When installed, Java handles the security issue by a Security-Manager-Object. The Security-Object protects predefined operations and enforces access controls over these operations. After the Security-Manager is installed, it cannot be removed or replaced by third parties.

Also the RMI framework provides a mechanism for clients protected by a firewall to communicate with a remote server across the Internet. However, crossing the client's firewall slows down the communication with the server; for this reason, the RMI framework uses a fast and sophisticated communication technique. This technique communicates directly with the server's port by using sockets, and is instantiated by the UnicastRemoteObject on the first call of the client to contact the server.

C. AREAS FOR FURTHER RESEARCH

As in every design and implementation phase, continuously researching and experimenting with the current technologies was necessary. When I look back upon various design decisions, there might have been a different and more efficient way to code and implement the software elements into the system. Since computer-technology is developing rapidly, expanding the current implementation in several areas in the near future will be necessary. At this point a program revision should be performed in the following areas:

1. **Streamline the Program Code:** The code for the project was written by using one of the available Rapid-Application-Development (RAD) tools for the Java programming language, namely the JBUILDER product. This tool is extremely powerful, providing a great deal of programming support for applications, and it has many customizable features, which are highly advantageous to a software developer. The installation was straightforward; the software automatically generated the appropriate classpath and setup. While these products, such as JBUILDER, are efficiently and technically reliable and supports the users during their software development, they also have some disadvantages compared to a pure Java software implementation, such as SUN's JDK1.x.x. coding environment. A great amount of additional code is generated when using the visual drag-and-drop component for creating a GUI. This produces larger class-files and a code unrefined for network implementation. A good practice would be to examine and to revise the code

in order to reduce the size of the class files. The intent of this time-consuming procedure would be to reduce the time for data downloading across the Internet and to improve the performance of the system.

2. **The Field Test of the System:** Without doubt, the true success of every system depends on the field test with real data. At that time, the developer can definitely evaluate the efficency of the system and survey the personnel's reactions to the system. Unfortunately, field testing and evaluating this system was beyond the scope of this thesis, but should be performed as a follow-on step, in conjunction with enhancing the code and the below mentioned implementation of a *Pick-List*.

3. **Extend the System and Implement Servlets:** Future upgrades of the system should include the integration of servlets. Servlets are loaded and executed by a Web server, and because they are executed on a server, security hazards are highly minimized. As previously demonstrated and explained, the Web browser does not communicate directly with a servlet; moreover, the servlet is invoked from the outside world only through the Web server. This provides a high standard of security, which can be further improved by protecting the Web server behind a secure firewall so that the connected servlet is secured as well. As a consequence, because servlets inherit the limitations provided by the Java security model, malicious hackers and intruders cannot exploit the government programs for their benefit.

4. **Create an Interactive Window (List Selection) for Data Input:** In the present design, the user is forced to update and to add data directly onto the

appropriate form. In order to improve the dialog between the user and the system, it would be beneficial to have a *Pick-list*, allowing the user to select the desired event by double-clicking the desired event with the mouse and the system would, in turn, add the event to the previously shown value on the sign-out sheet automatically. This would improve the user acceptance by speeding up the workflow and further reducing the possibility of user induced input errors.

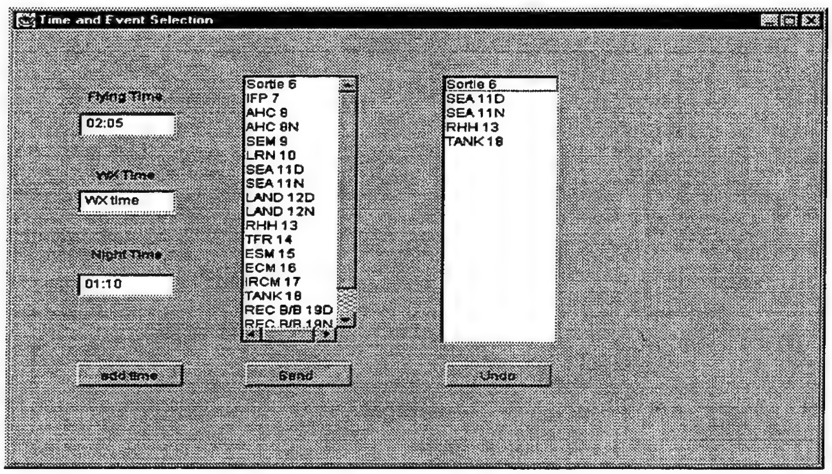


Figure 9.1: List Selection

APPENDIX A. SOURCE CODE USER

```
//--------------------------------------------------------------------
//--------------------------------------------------------------------
// File:        UserApp / dbSwing
// Name:        Enno F. Busch
// Date:        09-04-1999
// Company:     Bundeswehr (German Navy)
// Description: This class is used to instantiate the RMI user
//              application. It is the basis for the userFrame
//              class. It validates frames that have the preset size.
//--------------------------------------------------------------------

package Marine;

public class UserApp {
  boolean packFrame = false;

  public UserApp() {

    UserFrame frame = new UserFrame();
    if (packFrame) {
          frame.pack();
    }
    else {
      frame.validate();
    }

    frame.setVisible(true);
  }//end of public UserApp.

  /*
   * This method is used as a main method.
   */
  public static void main(String[] args) {
    new UserApp();
  }//end of main method.

}//end of public class UserApp
```

```
//-------------------------------------------------------------------
//-------------------------------------------------------------------
// File:        UserFrame / dbSwing
// Name:        Enno F. Busch
// Date:        09-14-1999
// Company:     Bundeswehr (German Navy)
// Description: This class is used to instantiate the RMI user
//              application.
//              This class is the basis for the userFrame class.
//              It validates frames that have the preset size.
//-------------------------------------------------------------------
package Marine;

import java.awt.*;
import java.awt.event.*;
import java.util.*;
import com.borland.dbswing.*;
import com.borland.dx.dataset.*;
import com.borland.dx.text.VariantFormatter;
import java.beans.*;
import javax.swing.*;

public class UserFrame extends DecoratedFrame {

  ResourceBundle res = Res.getBundle("Marine.Res");
  XYLayout xYLayout2 = new XYLayout();
  BevelPanel bevelPanel1 = new BevelPanel();
  com.borland.dx.dataset.TableDataSet tableDataSet1 =
            new com.borland.dx.dataset.TableDataSet();

  UserProvider clientProvider1 = new UserProvider();
  UserResolver clientResolver1 = new UserResolver();
  StatusBar statusBar = new StatusBar();
  BorderLayout borderLayout1 = new BorderLayout();
  Database database1 = new Database();
  QueryDataSet queryDataSet1 = new QueryDataSet();
  JdbNavToolBar jdbNavToolBar1 = new JdbNavToolBar();
  JdbStatusLabel jdbStatusLabel1 = new JdbStatusLabel();
  TableScrollPane tableScrollPane1 = new TableScrollPane();
  JdbTable jdbTable1 = new JdbTable();
  ParameterRow parameterRow1 = new ParameterRow();
  Column column1 = new Column();
  Column column2 = new Column();
  JdbTextField jdbTextField1 = new JdbTextField();
  JdbTextField jdbTextField2 = new JdbTextField();
  JdbLabel jdbLabel1 = new JdbLabel();
  JdbLabel jdbLabel2 = new JdbLabel();
  JLabel jLabel1 = new JLabel();
  StartupDialog startupDialog = new StartupDialog();
  DBPasswordPrompter dBPasswordPrompter1 = new  DBPasswordPrompter();
  JdbTextField jdbTextField3 = new JdbTextField();
  JdbTextField jdbTextField4 = new JdbTextField();
  JButton jButton1 = new JButton();
  Variant v = new Variant();
  String columnName = "Crew";
  String columnValue = "mol";
  VariantFormatter formatter;
```

```
  /*
   * This method is used to construct the frame for UserApp.
   */
  public UserFrame() {
    try  {
      jbInit();
    }
    catch(Exception e) {
      e.printStackTrace();
    }
  }

  /*
   * This method is used to initialize the componets.
   */
  private void jbInit() throws Exception  {

    this.setLayout(borderLayout1);
    this.setBackground(SystemColor.controlHighlight);
    this.setFont(new java.awt.Font("Dialog", 0, 16));
    this.setSize(new Dimension(1019, 742));
    this.setTitle(res.getString("RS_ClientTitle"));
    bevelPanel1.setLayout(xYLayout2);

    //startupDialog.setVisible(true);

    database1.setConnection(new
      com.borland.dx.sql.dataset.ConnectionDescriptor("jdbc:odbc:
      MFG3", "Busch", "Busch", true,
      "sun.jdbc.odbc.JdbcOdbcDriver"));
    database1.openConnection();

    queryDataSet1.setSort(new
      com.borland.dx.dataset.SortDescriptor("", new String[]
      {"CrewNo"}, new boolean[] {false, }, false, false, null));
    queryDataSet1.setQuery(new
      com.borland.dx.sql.dataset.QueryDescriptor(database1,
      "select * from table1, table2 WHERE
      table1.Crew=table2.CrewK AND CrewNo " +
    ">= :low_no and CrewNo <= :high_no", parameterRow1, true,
      Load.ASYNCHRONOUS));
    queryDataSet1.addRowFilterListener(new
      com.borland.dx.dataset.RowFilterListener() {

      public void filterRow(ReadRow readRow, RowFilterResponse
rowFilterResponse) throws DataSetException {
        queryDataSet1_filterRow(readRow, rowFilterResponse);
      }
    });
    jdbStatusLabel1.setText("jdbStatusLabel1");
    jdbTable1.setDataSet(queryDataSet1);
    column1.setColumnName("low_no");
    column1.setDataType(com.borland.dx.dataset.Variant.INT);
    column1.setDefault("4");
    column1.setServerColumnName("NewColumn1");
    column1.setSqlType(0);
```

```
column2.setColumnName("high_no");
column2.setDataType(com.borland.dx.dataset.Variant.INT);
column2.setDefault("4");
column2.setServerColumnName("NewColumn2");
column2.setSqlType(0);
tableDataSet1.setProvider(clientProvider1);
tableDataSet1.setResolver(clientResolver1);
//column definition
parameterRow1.setColumns(new Column[] {column1, column2});

//first textfield
jdbTextField1.setText("Min");
jdbTextField1.addKeyListener(new java.awt.event.KeyAdapter() {

  public void keyPressed(KeyEvent e) {
    jdbTextField1_keyPressed(e);
  }
});

//second textfield
jdbTextField2.setText("Max");
jdbTextField2.addKeyListener(new java.awt.event.KeyAdapter() {

  public void keyPressed(KeyEvent e) {
    jdbTextField2_keyPressed(e);
  }
});

jdbLabel1.setText("Min value,    1. Squadron   150 - 200");
jdbLabel2.setText("Max value,    2. Squadron   250 - 300");
jdbNavToolBar1.setToolTipText("Warning! Changed Events that   are
saved can not be restored!");

bevelPanel1.setBevelOuter(BevelPanel.LOWERED);
bevelPanel1.setSoft(true);
bevelPanel1.setToolTipText("");
jLabel1.setBackground(Color.gray);
jLabel1.setFont(new java.awt.Font("Dialog", 0, 26));
jLabel1.setText("MFG 2  Sign-Out Sheet");
dBPasswordPrompter1.setFrame(this);
dBPasswordPrompter1.setPassword("busch");
dBPasswordPrompter1.setUserName("busch");
jdbTextField3.setText("Crew");
jdbTextField4.setText(" ");
jButton1.setText("Selection");
jButton1.addActionListener(new
  java.awt.event.ActionListener() {

  public void actionPerformed(ActionEvent e) {
    jButton1_actionPerformed(e);
  }
});
this.add(statusBar, BorderLayout.NORTH);
this.add(bevelPanel1, BorderLayout.CENTER);
  bevelPanel1.add(tableScrollPanel, new XYConstraints(19, 92,
  936, 489));
```

```
    bevelPanel1.add(jdbNavToolBar1, new XYConstraints(18, 601,    707,
42));
    bevelPanel1.add(jdbLabel1, new XYConstraints(253, 14, 207,    26));
    bevelPanel1.add(jdbLabel2, new XYConstraints(253, 54, 205,    26));
    bevelPanel1.add(jdbTextField1, new XYConstraints(133, 12, 76,
      24));
    bevelPanel1.add(jdbTextField2, new XYConstraints(133, 53, 76,
      24));
    bevelPanel1.add(jdbTextField3, new XYConstraints(476, 12, 76,
      25));
    bevelPanel1.add(jdbTextField4, new XYConstraints(476, 53, 76,
      25));
    bevelPanel1.add(jButton1, new XYConstraints(580, 31, 107,     33));
    bevelPanel1.add(jLabel1, new XYConstraints(692, 8, 287, 46));
    tableScrollPanel.getViewport().add(jdbTable1, null);
    this.add(jdbStatusLabel1, BorderLayout.SOUTH);
  }

  /*
   * This method is used to process the first user-input from the
      textfield1 and
   * to refresh the dataset.
   */
  void jdbTextField1_keyPressed(KeyEvent e) {
    if (e.getKeyCode() == KeyEvent.VK_ENTER) {
       try {
          parameterRow1.setInt("low_no", Integer.parseInt
      (jdbTextField1.getText()));
          queryDataSet1.refresh();
       }
       catch (Exception ex) {
          ex.printStackTrace();
       }
    }//end of if
  }//end of jdbTextField1_keyPressed

  /*
   * This method is used to process the second user-input from    the
textfield2 and
   * to refresh the dataset.
   */
  void jdbTextField2_keyPressed(KeyEvent e) {
    if (e.getKeyCode() == KeyEvent.VK_ENTER) {
       try {
          parameterRow1.setInt("high_no", Integer.parseInt
      (jdbTextField2.getText()));
          queryDataSet1.refresh();
          System.out.println("Save");
       }
       catch (Exception ex) {
          ex.printStackTrace();
       }
    }//end of if
  }//end of jdbTextField2_keyPressed

  void queryDataSet1_filterRow(ReadRow readRow, RowFilterResponse
      rowFilterResponse) throws DataSetException {
```

```
      try {
         if (formatter == null || columnName == null ||       columnValue
== null ||
             columnName.length() == 0 || columnValue.length() ==
       0)

             rowFilterResponse.add();
         else {
            readRow.getVariant(columnName, v);
            String s = formatter.format(v);

            if (columnValue.equals(s))
               rowFilterResponse.add();
            else
               rowFilterResponse.ignore();
         }
      }
      catch (Exception e) {
         System.err.println("Filter failed");
      }
   }

   void jButton1_actionPerformed(ActionEvent e) {
      try {
         columnName =  jdbTextField3.getText();
         columnValue = jdbTextField4.getText();
         Column column = queryDataSet1.getColumn(columnName);
         formatter = column.getFormatter();

         queryDataSet1.refilter();
      }
      catch (Exception ex) {
         System.err.println("Filter failed");
      }
   }//end of jButton1_actionPerformed
}//end of class User frame
```

```
//---------------------------------------------------------------------
//---------------------------------------------------------------------
// File:        StartupDialog /dbSwing
// Name:        Enno F. Busch
// Date:        09-15-1999
// Company:     Bundeswehr (German Navy)
// Description: This class is used to instantiate the RMI user
//              application.
//---------------------------------------------------------------------

package Marine;

import java.awt.*;
import java.awt.event.*;
import com.borland.jbcl.control.*;
import com.borland.jbcl.layout.*;

public class StartupDialog extends DecoratedFrame {
  XYLayout xYLayout2 = new XYLayout();
  BevelPanel bevelPanel1 = new BevelPanel();
  StatusBar statusBar = new StatusBar();
  BorderLayout borderLayout1 = new BorderLayout();
  TextField textField1 = new TextField();
  TextField textField2 = new TextField();
  TextField textField3 = new TextField();
  Button button1 = new Button();
  Button button2 = new Button();
  Label label1 = new Label();
  Label label2 = new Label();
  Label label3 = new Label();
  Label label4 = new Label();

  //Construct the frame
  public StartupDialog() {
    try  {
      jbInit();
    }
    catch(Exception e) {
      e.printStackTrace();
    }
  }

  //Component initialization
  private void jbInit() throws Exception  {
    this.setLayout(borderLayout1);
    this.setSize(new Dimension(896, 563));
    this.setTitle("Startup Frame MFG 2");
    bevelPanel1.setLayout(xYLayout2);

    button1.setName("button1");
    button1.setLabel("Enter");
    button2.setLabel("Cancel");
    label1.setText("Enter Your Name");
    label2.setText("Enter your Password");
    label3.setText("Enter Database");
    label4.setFont(new java.awt.Font("Dialog", 0, 20));
    label4.setText("Welcome to Naval Airwing 2");
```

```
    this.add(statusBar, BorderLayout.SOUTH);
    this.add(bevelPanel1, BorderLayout.NORTH);
    bevelPanel1.add(button1, new XYConstraints(211, 306, 105, 25));
    bevelPanel1.add(button2, new XYConstraints(336, 306, 105, 25));
    bevelPanel1.add(textField1, new XYConstraints(58, 81, 127, 28));
    bevelPanel1.add(textField2, new XYConstraints(58, 131, 127, 28));
    bevelPanel1.add(textField3, new XYConstraints(58, 181, 127, 28));
    bevelPanel1.add(label1, new XYConstraints(209, 86, 126, 28));
    bevelPanel1.add(label2, new XYConstraints(209, 132, 126, 28));
    bevelPanel1.add(label3, new XYConstraints(209, 183, 126, 28));
    bevelPanel1.add(label4, new XYConstraints(98, 28, 269, 36));
  }

  void button1_actionPerformed(ActionEvent e) {
     dispose();
  }
}
```

```
//---------------------------------------------------------------------
//---------------------------------------------------------------------
// File:          Res.java /dbSwing
// Name:          Enno F. Busch
// Date:          09-10-1999
// Company:       Bundeswehr (German Navy)
// Description:   This class contains locale-specified objects. The
//                program loads all needed information from this
//                bundle, which is appropriate for the current user.
//---------------------------------------------------------------------
package Marine;

public class Res extends java.util.ListResourceBundle {

  static final Object[][] contents = new String[][]{
      { "RS_ServerTitle", "Marinefliegergeschwader 2 Server" },
      { "RS_CloseServer", "Shut Server Down" },
      { "RS_Hostname", "Hostname" },
      { "RS_NoRegistry", "Connection problem. Make sure to run
              rmiregistry from the \r\n jbuilder\\java\\bin " +
                                              "directory first." },
      { "RS_Requests", "Handled Client Requests:" },
      { "RS_Error", "Error" },
      { "RS_AlreadyRegistered", "Another server is already registered,
                                      Do you want to override that " +
                                                      "server?" },
      { "RS_ClientTitle", "Marinefliegergeschwader 2" },
      { "RS_NoServer", "No Data server avaliable, run DataServerApp
                                                        first." },
      { "RS_NoConnect", "Couldn\'t connect to DataServerApp on host "
                                                            }};

  public Object[][] getContents() {

    return contents;
  }
}//end of Resource class
```

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX B. SOURCE CODE SERVER

```
//--------------------------------------------------------------------------
//--------------------------------------------------------------------------
// File:        DataServerApp / dbSwing
// Name:        Enno F. Busch
// Date:        09-014-1999
// Company:     Bundeswehr (German Navy)
// Description: This class is used to instantiate the RMI server.
//
//--------------------------------------------------------------------------
package Marine;

import com.borland.jbcl.util.*;
import com.borland.jbcl.control.*;
import com.borland.jbcl.model.*;
import java.rmi.*;
import java.net.InetAddress;
import java.rmi.server.UnicastRemoteObject;
import java.util.*;

public class DataServerApp extends UnicastRemoteObject implements
                                                         CrewmemberApi {

   ResourceBundle res = Res.getBundle("Marine.Res");

  /*
   * This method is used as a constructor that displays the server's
frame.
   */
  public DataServerApp() throws RemoteException {
     frame = new DataServerFrame(this);
     frame.pack();
     frame.setVisible(true);
  }

  /*
   * This main-method is used to construct an instance variable and
calls the
   * bindToNaming Service.
   */
  public static void main(String[] args) {

     DataServerApp dataServerApp = null;
     try {
      dataServerApp = new DataServerApp();
      dataServerApp.bindToNamingService();
     }
     catch (Exception ex) {
       DataSetModel.handleException(ex, true);
       System.exit(1);
     }
  }
```

```
  /*
   * This method is used to handle the bindToNamingService. RMI
requires the actual hostName in the URL.
   */
  void bindToNamingService() throws Exception {
    InetAddress addr = InetAddress.getLocalHost();
    String localHost = addr.getHostName();
    serviceURL = "//" + localHost + "/DataServerApp";
    try {
      Naming.bind(serviceURL, this);
      frame.textControl2.setText(res.getString("RS_Hostname")+":
                                                        "+localHost);
    }
    catch (java.rmi.AlreadyBoundException ex) {
      MessageDialog dlg = new MessageDialog(frame,
      res.getString("RS_Error"), res.getString("RS_AlreadyRegistered"),
                                           ButtonDialog.YES_NO);
      dlg.setVisible(true);
      if (dlg.getResult() == ButtonDialog.YES) {
                                 Naming.rebind(serviceURL, this);
      }
    }
    catch (java.rmi.ConnectException ex) {
      ExceptionChain chain = new ExceptionChain();
      chain.append(ex);
      throw new DataSetException(DataSetException.GENERIC_ERROR,
                          res.getString("RS_NoRegistry"), chain);
    }
  }

  /*
   * This method is used to unbind the service from the RMI. It is
called as a
   * result of closing the frame from UserFrame.java.
   */
  void closing() {
    try {
      Naming.unbind(serviceURL);
    }
    catch (Exception ex) {
    }
  }

  // Constants for the database connection properties.
  final String  url          = "jdbc:odbc:MFG3";
  final String  driver       = "sun.jdbc.odbc.JdbcOdbcDriver";
  final String  username     = "Busch";
  final String  password     = " ";
  final String  queryText    = "select * from table1, table2 ";

 public DataSetData provideCrewmemberData() throws RemoteException,
                                               DataSetException {
    frame.incrementUsage();
    Database     db   = new Database();
    QueryDataSet qds  = new QueryDataSet();
```

```
    try {
      db.setConnection(new ConnectionDescriptor(url, username,
                                             password, false, driver));
      //db.openConnection();
      qds.setQuery(new QueryDescriptor(db, queryText, null, true,
                                                         Load.ALL));
      qds.open();
      DataSetData  data = DataSetData.extractDataSet(qds);
      return data;
    }
    finally {
      qds.close();
      db.closeConnection();
    }
  }

  /*
   * This method is used to resolve the Crewmember changes. It also
provides the
   * implementation of the Crewmember interface.
   * First, the MetaData are called; hence, the MaxRows are specified
to 0.
   * Second, the QueryDataSet is filled with data.
   * Third, all errors are returned as another DataSet instance.
   */
  public DataSetData resolveCrewmemberChanges(DataSetData changes)
throws RemoteException, DataSetException {
    frame.incrementUsage();
    Database       db   = new Database();
    QueryDataSet   qds  = new QueryDataSet();
    QueryResolver  qr   = new QueryResolver();
    ResolverHelp   rh   = new ResolverHelp();
    try {
      db.setConnection(new ConnectionDescriptor(url, username,
                                          password, false, driver));
      qds.setQuery(new QueryDescriptor(db, queryText, null, true,
                                                      Load.ALL));
      qr.setDatabase(db);
      qr.addResolverListener(rh);
      qds.setResolver(qr);
      qds.setMaxRows(0);
      qds.open();
      changes.loadDataSet(qds);
      qds.saveChanges();
      return rh.getErrors();
    }
    catch (java.util.TooManyListenersException ex) {
                    throw new DataSetException(ex.getMessage());
    }
    finally {
       qds.close();
       db.closeConnection();
    }
  }

  // Keep the service URL for the closing method.
  private String serviceURL;
```

```
  // Keep the frame around for various messages.
  private DataServerFrame frame;
}

/*
 * This class is used as a resolver for errors. It is told to ignore
all errors.
 * All data changes that does not cause any problems are saved and
tranferred to
 * the database. Errors are logged by this class and stored into a
dataset and
 * returned to the User Application.
 */
class ResolverHelp extends ResolverAdapter {

  /*
   * This method is used to implement "insertError" into
ResolverListener.
   */
  public void insertError(DataSet dataSet, ReadWriteRow row,
           DataSetException ex, ErrorResponse response) throws
                                                   DataSetException {
    handleError(dataSet, ex, response);
  }

  /*
   *This method is used to implement "deleteError" into
ResolverListener.
   */
  public void deleteError(DataSet dataSet, ReadWriteRow row,
           DataSetException ex, ErrorResponse response) throws
                                                   DataSetException {
    handleError(dataSet, ex, response);
  }

  /*
   * This method is used to implement "updateError" into
ResolverListener.
   */
  public void updateError(DataSet dataSet, ReadWriteRow row, ReadRow
      oldRow, ReadWriteRow updRow, DataSetException ex, ErrorResponse
                              response)  throws DataSetException {
    handleError(dataSet, ex, response);
  }

  /*
   * This method is used to return a DataSetData with the accumulated
errors.
   */
  DataSetData getErrors() throws DataSetException {
                        return errorDataSet == null ? null :
                              DataSetData.extractDataSet(errorDataSet);
  }

  /*
   * This method is used to handle errors. Errors are logged into the
```

```
   * Error DataSet. The method loggs for each error the internal row in
the
   * dataset inside the UserApp, which causes the problem. The
information is
   * stored by the DataSetDat.extractDataSetChanges method.
   */
  private void handleError(DataSet dataSet, DataSetException ex,
                  ErrorResponse response) throws DataSetException {
    createErrorDataSet();
    errorDataSet.insertRow(false);
    errorDataSet.setLong(internalRow,dataSet.getLong(internalRow));
    errorDataSet.setObject(exception,ex);
    errorDataSet.post();
    response.ignore();
  }

  /*
   * This method is used to create a dataSet and to collect the errors.
   */
  private void createErrorDataSet() throws DataSetException {
    if (errorDataSet != null)
      return;

      errorDataSet = new TableDataSet();
      errorDataSet.setColumns(new Column[]{new
                  Column(internalRow,internalRow,Variant.LONG),
                                        new
                        Column(exception,exception,Variant.OBJECT),
                                        });
      errorDataSet.open();
  }

  TableDataSet errorDataSet;
  static final String internalRow = "INTERNALROW";
  static final String exception   = "EXCEPTION";
}
```

```java
//----------------------------------------------------------------------
//----------------------------------------------------------------------
// File:        DataServerFrame / dbSwing
// Name:        Enno F. Busch
// Date:        09-09-1999
// Company:     Bundeswehr (German Navy)
// Description: This class is used to display the frame for the
//              DataServerApp.
//
//----------------------------------------------------------------------
package Marine;

import java.awt.*;
import java.awt.event.*;
import com.borland.jbcl.layout.*;
import com.borland.jbcl.control.*;
import java.util.*;


public class DataServerFrame extends DecoratedFrame {

  ResourceBundle res = Res.getBundle("Marine.Res");
  com.borland.jbcl.control.ButtonControl buttonControl1 = new
      com.borland.jbcl.control.ButtonControl();
  com.borland.jbcl.control.TextControl textControl1 = new
      com.borland.jbcl.control.TextControl();
  com.borland.jbcl.control.TextControl textControl3 = new
      com.borland.jbcl.control.TextControl();
  java.awt.GridBagLayout gridBagLayout1 = new java.awt.GridBagLayout();

  public DataServerFrame(DataServerApp app) {
    this.app = app;
    try {
      jbInit();
    }
    catch (Exception e) {
      e.printStackTrace();
    }
  }
  private void jbInit() throws Exception{
    this.setResizable(false);
    this.setSize(new Dimension(514, 429));
    this.setTitle(res.getString("RS_ServerTitle"));
    this.addWindowListener(new java.awt.event.WindowAdapter() {
      public void windowClosing(WindowEvent e) {
        this_windowClosing(e);
      }
    });
    buttonControl1.setBackground(Color.green);
    buttonControl1.setFont(new java.awt.Font("Dialog", 1, 22));
    buttonControl1.setForeground(Color.red);
    buttonControl1.setLabel(res.getString("RS_CloseServer"));
    buttonControl1.addActionListener(new
      java.awt.event.ActionListener() {
      public void actionPerformed(ActionEvent e) {
        buttonControl1_actionPerformed(e);
      }
```

```
    });
    textControl1.setBackground(SystemColor.controlShadow);
    textControl1.setEdgeColor(Color.red);
    textControl1.setFont(new java.awt.Font("Dialog", 1, 22));
    textControl1.setText(res.getString("RS_Requests"));

      textControl3.setAlignment(com.borland.dx.text.Alignment.CENTER |
      com.borland.dx.text.Alignment.MIDDLE);
    textControl3.setFont(new java.awt.Font("Dialog", 1, 38));
    textControl3.setText("0");
    this.setLayout(gridBagLayout1);
          this.add(buttonControl1, new GridBagConstraints(0, 3, 3, 1,
      0.0, 0.0
            ,GridBagConstraints.CENTER, GridBagConstraints.NONE, new
Insets(20, 20, 20, 20), 53, 45));
    this.add(textControl3, new GridBagConstraints(0, 2, 3, 1, 1.0, 1.0
            ,GridBagConstraints.CENTER, GridBagConstraints.BOTH, new
Insets(0, 20, 0, 20), 0, 0));
    this.add(textControl2, new GridBagConstraints(0, 0, 3, 1, 1.0, 1.0
            ,GridBagConstraints.CENTER, GridBagConstraints.BOTH, new
Insets(20, 20, 5, 20), 0, 0));
    this.add(textControl1, new GridBagConstraints(0, 1, 3, 1, 1.0, 1.0
            ,GridBagConstraints.NORTHEAST, GridBagConstraints.BOTH, new
Insets(15, 40, 5, 0), 0, 0));
  }

  void incrementUsage() {
    textControl3.setText(Integer.toString(++usage));
  }

  void buttonControl1_actionPerformed(ActionEvent e) {
    app.closing();
    System.exit(1);
  }

  void this_windowClosing(WindowEvent e) {
    app.closing();
  }

  private DataServerApp app;
  private int usage = 0;
  com.borland.jbcl.control.TextControl textControl2 = new
com.borland.jbcl.control.TextControl();
}
```

```
//--------------------------------------------------------------------------
//--------------------------------------------------------------------------
// File:        DataServerApp_Skel
// Name:        Enno F. Busch
// Date:        09-11-1999
// Company:     Bundeswehr (German Navy)
// Description: This skel class is instantiated by rmi. After compile
//              contents may change without any notice.
//--------------------------------------------------------------------------
package Marine;

public final class DataServerApp_Skel
    implements java.rmi.server.Skeleton
{
    private static final java.rmi.server.Operation[] operations = {
      new java.rmi.server.Operation("com.borland.dx.dataset.DataSetData
            provideCrewmemberData()"),
      new java.rmi.server.Operation("com.borland.dx.dataset.DataSetData
      resolveCrewmemberChanges(com.borland.dx.dataset.DataSetData)")
    };

    private static final long interfaceHash = -8746639934740811853L;

    public java.rmi.server.Operation[] getOperations() {
      return (java.rmi.server.Operation[]) operations.clone();
    }

    public void dispatch(java.rmi.Remote obj,
      java.rmi.server.RemoteCall call, int opnum, long hash)
                                    throws java.lang.Exception
    {
      if (opnum < 0) {
          if (hash == -5281998362710437 72L) {
            opnum = 0;
          } else if (hash == 7264815700894346221L) {
            opnum = 1;
          } else {
            throw new java.rmi.UnmarshalException("invalid method
                                                            hash");
          }
      } else {
          if (hash != interfaceHash)
            throw new
      java.rmi.server.SkeletonMismatchException("interface hash
      mismatch");
      }

      Marine.DataServerApp server = (Marine.DataServerApp) obj;
            switch (opnum) {
      case 0: // provideCrewmemberData()
      {
          call.releaseInputStream();
          com.borland.dx.dataset.DataSetData $result =
            server.provideCrewmemberData();
          try {
            java.io.ObjectOutput out = call.getResultStream(true);
            out.writeObject($result);
```

```
            } catch (java.io.IOException e) {
              throw new java.rmi.MarshalException("error marshalling
              return", e);
            }
            break;
        }

        case 1: // resolveCrewmemberChanges(DataSetData)
        {
            com.borland.dx.dataset.DataSetData $param_DataSetData_1;
            try {
              java.io.ObjectInput in = call.getInputStream();
              $param_DataSetData_1 = (com.borland.dx.dataset.DataSetData)
              in.readObject();
            } catch (java.io.IOException e) {
              throw new java.rmi.UnmarshalException("error unmarshalling
              arguments", e);
            } catch (java.lang.ClassNotFoundException e) {
              throw new java.rmi.UnmarshalException("error unmarshalling
              arguments", e);
            } finally {
              call.releaseInputStream();
            }
            com.borland.dx.dataset.DataSetData $result =
              server.resolveCrewmemberChanges($param_DataSetData_1);
            try {
              java.io.ObjectOutput out = call.getResultStream(true);
              out.writeObject($result);
            } catch (java.io.IOException e) {
              throw new java.rmi.MarshalException("error marshalling
              return", e);
            }
            break;
        }

        default:
            throw new java.rmi.UnmarshalException("invalid method
              number");
        }
    }
}
```

```
//-----------------------------------------------------------------------
//-----------------------------------------------------------------------
// File:          DataServerApp_Stub
// Name:          Enno F. Busch
// Date:          09-09-1999
// Company:     Bundeswehr (German Navy)
// Description: This stub class is instantiated by rmi. After compile
//               contents may change without any notice.
//
//-----------------------------------------------------------------------
package Marine;

public final class DataServerApp_Stub
    extends java.rmi.server.RemoteStub
                   implements Marine.CrewmemberApi, java.rmi.Remote
{
    private static final java.rmi.server.Operation[] operations = {
      new java.rmi.server.Operation("com.borland.dx.dataset.DataSetData
            provideCrewmemberData()"),
      new java.rmi.server.Operation("com.borland.dx.dataset.DataSetData

      resolveCrewmemberChanges(com.borland.dx.dataset.DataSetData)")
    };

    private static final long interfaceHash = -8746639934740811853L;

    private static final long serialVersionUID = 2;

    private static boolean useNewInvoke;
    private static java.lang.reflect.Method
      $method_provideCrewmemberData_0;
    private static java.lang.reflect.Method
      $method_resolveCrewmemberChanges_1;

    static {
      try {
          java.rmi.server.RemoteRef.class.getMethod("invoke",
            new java.lang.Class[] {
                java.rmi.Remote.class,
                java.lang.reflect.Method.class,
                java.lang.Object[].class,
                long.class
            });
          useNewInvoke = true;
          $method_provideCrewmemberData_0 =
      Marine.CrewmemberApi.class.getMethod("provideCrewmemberData", new
      java.lang.Class[] {});
          $method_resolveCrewmemberChanges_1 =
      Marine.CrewmemberApi.class.getMethod("resolveCrewmemberChanges",
      new java.lang.Class[]
      {com.borland.dx.dataset.DataSetData.class});
      } catch (java.lang.NoSuchMethodException e) {
          useNewInvoke = false;
      }
    }
```

```
/*
 * This methods is used to provide the constructors.
 */
public DataServerApp_Stub() {
       super();
}
public DataServerApp_Stub(java.rmi.server.RemoteRef ref) {
       super(ref);
}

// methods from remote interfaces

// implementation of provideCrewmemberData()
public com.borland.dx.dataset.DataSetData provideCrewmemberData()
  throws com.borland.dx.dataset.DataSetException,
  java.rmi.RemoteException
{
  try {
      if (useNewInvoke) {
        Object $result = ref.invoke(this,
  $method_provideCrewmemberData_0, null, -5281998362710437772L);
        return ((com.borland.dx.dataset.DataSetData) $result);
      } else {
        java.rmi.server.RemoteCall call =
  ref.newCall((java.rmi.server.RemoteObject) this, operations, 0,
  interfaceHash);
        ref.invoke(call);
        com.borland.dx.dataset.DataSetData $result;
        try {
            java.io.ObjectInput in = call.getInputStream();
            $result = (com.borland.dx.dataset.DataSetData)
                                                in.readObject();
        } catch (java.io.IOException e) {
            throw new java.rmi.UnmarshalException("error
                                   unmarshalling return", e);
        } catch (java.lang.ClassNotFoundException e) {
            throw new java.rmi.UnmarshalException("error
                                   unmarshalling return", e);
        } finally {
            ref.done(call);
        }
        return $result;
      }
  } catch (java.lang.RuntimeException e) {
      throw e;
  } catch (java.rmi.RemoteException e) {
      throw e;
  } catch (com.borland.dx.dataset.DataSetException e) {
      throw e;
  } catch (java.lang.Exception e) {
      throw new java.rmi.UnexpectedException("undeclared checked
  exception", e);
  }
}
```

```
// implementation of resolveCrewmemberChanges(DataSetData)
public com.borland.dx.dataset.DataSetData
  resolveCrewmemberChanges(com.borland.dx.dataset.DataSetData
  $param_DataSetData_1)
  throws com.borland.dx.dataset.DataSetException,
  java.rmi.RemoteException
{
  try {
      if (useNewInvoke) {
        Object $result = ref.invoke(this,
  $method_resolveCrewmemberChanges_1, new java.lang.Object[]
  {$param_DataSetData_1}, 7264815700894346221L);
        return ((com.borland.dx.dataset.DataSetData) $result);
      } else {
        java.rmi.server.RemoteCall call =
  ref.newCall((java.rmi.server.RemoteObject) this, operations, 1,
  interfaceHash);
        try {
            java.io.ObjectOutput out = call.getOutputStream();
            out.writeObject($param_DataSetData_1);
        } catch (java.io.IOException e) {
            throw new java.rmi.MarshalException("error marshalling
  arguments", e);
        }
        ref.invoke(call);
        com.borland.dx.dataset.DataSetData $result;
        try {
            java.io.ObjectInput in = call.getInputStream();
            $result = (com.borland.dx.dataset.DataSetData)
  in.readObject();
        } catch (java.io.IOException e) {
            throw new java.rmi.UnmarshalException("error
  unmarshalling return", e);
        } catch (java.lang.ClassNotFoundException e) {
            throw new java.rmi.UnmarshalException("error
  unmarshalling return", e);
        } finally {
            ref.done(call);
        }
        return $result;
      }
  } catch (java.lang.RuntimeException e) {
      throw e;
  } catch (java.rmi.RemoteException e) {
      throw e;
  } catch (com.borland.dx.dataset.DataSetException e) {
      throw e;
  } catch (java.lang.Exception e) {
      throw new java.rmi.UnexpectedException("undeclared checked
  exception", e);
  }
}
}
```

```
//-----------------------------------------------------------------------
//-----------------------------------------------------------------------
// File:        UserResolver / dbSwing
// Name:        Enno F. Busch
// Date:        09-11-1999
// Company:     Bundeswehr (German Navy)
// Description: This class is used as an implementation of a
//              resolver. The program look up the DataServerApp program
//              on the host that is specified by the hostName property.
//              In a second step any changes are extracted in a DataSet
//              instance.
//              In a third step a remote call is initiated, and if any
//              errors are encountered, they are handled accordingly.
//
//-----------------------------------------------------------------------

package Marine;

import java.rmi.*;
import java.util.*;

class UserResolver extends Resolver {

  ResourceBundle res = Res.getBundle("Marine.Res");

  public void resolveData(DataSet dataSet) throws DataSetException {
    try {
      String         serverName = "//" + hostName + "/DataServerApp";
      EmployeeApi    server   = (EmployeeApi)Naming.lookup(serverName);
      ProviderHelp.startResolution(dataSet.getStorageDataSet(), true);

      DataSetData changes = DataSetData.extractDataSetChanges(dataSet);
      DataSetData errors  = server.resolveEmployeeChanges(changes);
      if (errors != null)
         handleErrors(dataSet,errors);
         dataSet.resetPendingStatus(true);
    }
    catch (DataSetException ex) {
       dataSet.resetPendingStatus(false);
       throw ex;
    }
    catch (NotBoundException ex) {
      dataSet.resetPendingStatus(false);
      throw new DataSetException(res.getString("RS_NoServer"));
    }
    catch (Exception ex) {
      dataSet.resetPendingStatus(false);
      String error = ex.getMessage();
      if (error == null)
        error = res.getString("RS_NoConnect") + hostName;
        throw new DataSetException(error);
    }
    finally {
      ProviderHelp.endResolution(dataSet.getStorageDataSet());
    }
  }
```

```
  /*
   * This method handles the errors. It loops through all the errors
   * reported, and resets the status bits to not resolved for each row
   * that caused an error. Then we go to the first reported error, and
   * let the provider fail with that error.
   */
  private void handleErrors(DataSet dataSet, DataSetData errors)
    throws DataSetException
  {
    TableDataSet errorDataSet = new TableDataSet();
    errors.loadDataSet(errorDataSet);
    long internalRow;
    errorDataSet.first();
    do {
      internalRow = errorDataSet.getLong(0);
      dataSet.resetPendingStatus(internalRow,false);
    } while (errorDataSet.next());

    dataSet.resetPendingStatus(true);

    errorDataSet.first();
    internalRow = errorDataSet.getLong(0);
    dataSet.goToInternalRow(internalRow);
    DataSetException ex = (DataSetException)errorDataSet.getObject(1);
    throw ex;
  }

  /*
   * This method is used to locate the "DataServerApp"
   * service through RMI. By default the program looks at the local
   * machine.
   */
  public String getHostName() {
    return hostName;
  }

  public void setHostName(String hostName) {
    this.hostName = hostName;
  }

  private String hostName = "localhost";
}
```

```
//---------------------------------------------------------------------
//---------------------------------------------------------------------
// File:        UserProvider / dbSwing
// Name:        Enno F. Busch
// Date:        09-11-1999
// Company:     Bundeswehr (German Navy)
// Description: This class is used as an implementation of a provider.
//              The program looks up the DataServerApp program
//              on the host that is specified by the hostName property.
//              In a second step the program performs a remote call to
//              provideCrewmemberData and laoads the DataSet with data.
//---------------------------------------------------------------------
package Marine;

import java.rmi.*;
import java.util.*;

class UserProvider extends Provider {

  ResourceBundle res = Res.getBundle("Marine.Res");

  public void provideData(StorageDataSet dataSet, boolean toOpen)
                                           throws DataSetException {
    try {
      String       serverName = "//" + hostName + "/DataServerApp";
      CrewmemberApi server = (CrewmemberApi)Naming.lookup(serverName);
      DataSetData data = server.provideCrewmemberData();
      dataSet.empty();
      data.loadDataSet(dataSet);
    }
    catch (NotBoundException ex) {
           throw new DataSetException(res.getString("RS_NoServer"));
    }
    catch (DataSetException ex) {
      throw ex;
    }
    catch (Exception ex) {
      String error = ex.getMessage();
         if (error == null)
            error = res.getString("RS_NoConnect")+hostName;
            throw new DataSetException(error);
    }
  }

  /*
   * This method is used to locate the "DataServerApp" service through
   * RMI. By default it looks to the local machine.
   */
  public String getHostName() {
    return hostName;
  }

  public void setHostName(String hostName) {
    this.hostName = hostName;
  }
  private String hostName = "localhost";
}
```

```
//------------------------------------------------------------------------
//------------------------------------------------------------------------
// File:        CrewmemberApi /dbSwing
// Name:        Enno F. Busch
// Date:        08-14-1999
// Company:     Bundeswehr (German Navy)
// Description: This method is used to define the methods to be used
//              remotely.
//------------------------------------------------------------------------

package Marine;

import com.borland.dx.dataset.DataSetData;
import com.borland.dx.dataset.DataSetException;
import java.rmi.Remote;
import java.rmi.RemoteException;


 /*
  * This method is used to define the methods to be used remotely.
  */
public interface CrewmemberApi extends Remote {

DataSetData provideCrewmemberData() throws RemoteException,
                                                  DataSetException;
DataSetData resolveCrewmemberChanges(DataSetData changes) throws
                                   RemoteException, DataSetException;

}//end of CrewmemberApi
```

LIST OF REFERENCES

[AL99] Akbay, M., Lewis, S., *Design and Implementation of an Enterprise Information System Utilizing a Component Based Three Tier Client-Server Database System*, Thesis, NPS Monterey, CA, March 1999.

[ARMS98] Armstrong, Eric, *Jbuilder2 Bible*, IDG Books Worldwide, Inc., 1998.

[BROO95] Brooks, Frederick, *The Mythical Man-Month*, Addison Wesley Longman, Inc., 1995.

[FLAN97] Flanagan, David, *JAVA IN A NUTSHELL, A Desktop Quick Reference*, O'Reilly & Associates, Inc., 1997.

[HCF97] Hamilton, Graham and Cattell, Rick and Fisher, Maydene, *JDBC Database Access with Java, A Tutorial and Annotated Reference*, The Java Series, Addison Wesley Longman, Inc., 1997.

[HM99] Held, J., Mingo, F., *Automating the Aviation Command Safety Assessment Survey as an Enterprise Information System*, Thesis, NPS Monterey, CA, March 1999.

[JAVA99] *The Source for Java Technology*, http://www.java.sun.com

[JAVA99a] *The Source for Java Technology*, http://java.sun.com/products

[LARM97] Larman, Craig, *Applying UML and Patterns, An Introduction to Object-Oriented Analysis and Design*, Prentice-Hall, Inc., 1997.

[MICR97] Microsoft Press, *Microsoft Access 97, At a Glance*, Visual Reference, Microsoft Press, 1997.

[SIPL98] Siple, Matthew, *The Complete Guide to JAVA Database Programming*, Computing McGraw-Hill, 1998.

[SISH94] Singhal, Mukesh and Shivaratri, Niranjan, *Advanced Concepts in Operating Systems*, McGraw-Hill Series in Computer Science, McGraw-Hill, Inc. 1994.

[SUND98] Sunderraman, Rajshekhar, *Oracle Programming, A PRIMER*, Addison-Wesley Longman, Inc. 1998.

[TANE97] Tanenbaum, A. and Woodhull, A., *Operating Systems: Design and Implementation*, Upper Saddle River, NJ: Prentice Hall, 1997.

[TARE85] Tanenbaum, A. and vanRenesse, R., *Distributed Operating System,* Computing Surveys ACM, vol. 17, no.4, Dec. 1985, pp. 419-470.

[WANG98] Wang, Paul S., *JAVA with Object-Oriented Programming and World Wide Web Applications*, PWS Publishing, 1998.

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center 2
   8725 John J. Kingman Rd., Ste 0944
   Fort Belvoir, VA 22060-6218

2. Dudley Knox Library 2
   Naval Postgraduate School
   411 Dyer Road
   Monterey, CA 93943-5101

3. Commanding Officer 1
   KdoMFueSys
   Wibbelhofstr. 3
   26384 Wilhelmshaven, Germany

4. Chairman, Code CS 1
   Naval Postgraduate School
   Monterey, CA 93943-5101

5. Dr. C. Thomas Wu, Code CS/Wu 1
   Naval Postgraduate School
   Monterey, CA 93943-5100

6. Michael Capps, Code CS/Ca 1
   Naval Postgraduate School
   Monterey,CA 93943-5100

7. CDR (GNY) Enno F. Busch 2
   KdoMFueSys, Gruppe II
   Wibbelhofstr. 3
   26384 Wilhelmshaven, Germany